Parallel programming models and run-time system support for interactive multimedia applications

Paul B. Beskow

To my family

Abstract

The computational demands of interactive multimedia applications are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multi-core hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counter-part. For this reason a number of parallel programming models, tools and techniques exist to alleviate the cognitive load placed on the developer, where a number of these solutions allow a developer to think sequentially, yet benefit from parallel and distributed execution. An inherit limitation in a number of these existing solutions is their inability to express arbitrarily complex workloads. The dependency graphs of these approaches, which express the relationship between computation and communication as vertices and edges, are often limited to directed acyclic graphs or even pre-determined stages. Furthermore, these existing solutions are frequently only capable of expressing a subset of the available parallelization domains. This effectively limits these solutions applicability to video encoding and other multimedia algorithms that depend on iterative execution and are capable of expressing a number of forms of parallelism. Through the development of Ginnungagap and evaluation of Nornir, and related work, we have extracted concepts for the formulation of a new high-level programming model and subsequent run-time system, called P2G, which is capable of supporting the emerging domain of interactive multimedia applications. With P2G we improve on existing tools and techniques with the support of arbitrarily complex dependency graphs with cycles, branches and deadlines, and provide support for data, task, pipeline and nested parallelism. This is achieved in P2G through the definition a high-level programming model, implemented through a novel kernel language that is designed to minimize the development effort otherwise placed on the developer by automating key components for parallelization and distribution, such as partitioning, scheduling, agglomeration and synchronization. To demonstrate the feasibility of our solution, we have implemented a proof-of-concept run-time system that demonstrates the applicability and scalability of the designed solution.

Acknowledgments

I would like to sincerely thank my supervisors, Prof. Carsten Griwodz and Prof. Pål Halvorsen for their support and dedication. Without their help and encouragement this thesis would not have been possible. It has been an honor to work so closely with such dedicated researchers. The work summarized in this thesis represents only a small amount of the knowledge I have gained while working with them.

To all the great colleagues I have had up through the years; Håvard Espeland, who has contributed to this thesis work at a number of levels, and has greatly contributed to an engaging work place. To my office mates, Håkon Stensland and Peng-peng Ni, for providing countless hours of great conversations. For consistently providing a great work environment; Andreas Petlund, Kristian Evensen, Ragnhild Eg, Ahmed Elmokashfi, Zeljko Vrba, Preben N. Olsen, Espen A. Kristiansen and Ståle Kristoffersen.

To all the great guys and gals that have had a home in the lab at Simula up through the years; Alexander Ottesen, Martin Myrseth, Kjetil Ree, Magne Eimot, Tonje Fredrikson, Bjørnar Snoksrud and Torkild Retvedt.

To all the people at Simula and the Networks and Distributed Systems group at IFI who have contributed to inspiring working conditions.

As is often the case, those who have felt the sacrifices I have made most keenly are those closest to me. I would therefore like to express my sincere gratitude to my family and closest friends for their continual support and dedication up through the years. A PhD will teach you a number of things, but the most valuable knowledge I have gained comes from the values you have imparted me with. Without those values it would have been impossible for me to ever start on this project or even complete it. To my parents, for teaching me the value of hard-work and patience, and always ensuring that I remained focused. There have been moments of doubt and without your support I would not have been where I am today. Your continual encouragement, support and love has been invaluable.

To the best brother and sisters one could ask for, to Anna, for teaching me to relax (and how to write), to Sara, for teaching me to be creative, to Erik, for teaching me the value of dedication. You have all been very patient with me up through the years, words cannot describe how thankful I am for this.

To my very good friend Yngve Austvoll for providing countless distractions in my spare time; some day I will become a better board game player then you.

And last, but not least, I would like to thank my girlfriend Marit, simply because you are you.

Contents

I Overview **1**

1 Introduction **3**
1.1 Application Domain 5
1.2 Heterogeneous, distributed hardware resources 6
1.3 Range of parallel programming solutions 7
1.4 Motivation 8
1.5 Research objectives 8
1.6 Scope and delimitation 9
1.7 Research method 10
1.8 Main contributions 10
1.8.1 Deferred decisions 11
1.8.2 Decoupling 11
1.8.3 Distributed name service 11
1.8.4 Migration or resource locality 12
1.8.5 Explicit and implicit dependency graphs 12
1.8.6 Compositionality 12
1.8.7 Determinism 13
1.8.8 Named state 13
1.9 Disposition of the thesis 13

2 Parallelization opportunities in interactive multimedia applications **15**
2.1 Decomposition domains 16
2.1.1 Partitioning 17
2.2 Workloads 18
2.2.1 Event processing 18
2.2.2 *k*-means clustering algorithm 19
2.2.3 H.264/AVC 22
2.3 Conclusion 33

3 Parallel Programming Model **35**
3.1 Designing parallel algorithms 35
3.2 Qualities 37
3.2.1 System 37
3.2.2 Maintenance 38

3.2.3 Development 39
3.2.4 Summary 41
3.3 Concepts 41
3.3.1 Named state 43
3.3.2 Determinism 44
3.3.3 Forms of parallelism 46
3.3.4 Iterative behavior 47
3.3.5 Data centric execution 49
3.3.6 Functional programming model 50
3.4 Delivery 52
3.4.1 Automatic parallelization 53
3.4.2 Library 53
3.4.3 New programming language 53
3.4.4 Coordination language 54
3.5 Conclusion 54

4 Parallel run-time system **57**
4.1 Relation to high-level programming model 58
4.1.1 Decoupling 58
4.1.2 Representation (processing graphs) 60
4.2 Run-time system fundamentals 65
4.2.1 Profiling 65
4.2.2 Scalable parallel execution 65
4.2.3 Scheduling tasks 68
4.3 Conclusion 70

5 Communication model **73**
5.1 Communication modes 73
5.2 Name service 74
5.3 State communication semantics 75
5.3.1 Implicit dissemination of state 76
5.3.2 Explicit dissemination of state 78
5.4 Conclusion 81

6 Papers and contributions **83**
6.1 Overview of research papers 83
6.2 Ginnungagap 83
Paper I: Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State 84
Paper II: Latency Reduction by Dynamic Core Selection and Partial Migration of Game State 84
Paper III: Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction 85
Paper IV: The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency 85
Paper V: Reducing game latency by migration, core-selection and TCP modifications 85

6.3 Nornir 86
Paper VI: Kahn process networks are a flexible alternative to MapReduce . . . 86
Paper VII: The Nornir run-time system for parallel programs using Kahn process networks 87
Paper VIII: The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce 87
6.4 P2G 87
Paper IX: Distributed Real-Time Processing of Multimedia Data with the P2G Framework 88
Paper X: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data 88
Paper XI: Processing of Multimedia Data using the P2G Framework 88
6.5 Lessons learned 89
6.5.1 Ginnungagap 89
6.5.2 Nornir 90
6.5.3 P2G 91
6.5.4 Overall 92

7 Conclusion **95**
7.1 Fulfillment of research objectives 95
7.1.1 RO1: Workloads 95
7.1.2 RO2: Abstractions 96
7.1.3 RO3: Execution 97
7.1.4 RO4: Validation 98
7.2 Critical assessments 99
7.3 Future work 100

II Research Papers 109

Paper I: Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State **111**

Paper II: Latency Reduction by Dynamic Core Selection and Partial Migration of Game State **123**

Paper III: Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction **131**

Paper IV: The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency **139**

Paper V: Reducing game latency by migration, core-selection and TCP modifications **165**

Paper VI: Kahn process networks are a flexible alternative to MapReduce **187**

Paper VII: The Nornir run-time system for parallel programs using Kahn process networks **199**

Paper VIII: The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce **209**

Paper IX: Distributed Real-Time Processing of Multimedia Data with the P2G Framework **239**

Paper X: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data **245**

Paper XI: Processing of Multimedia Data using the P2G Framework **259**

Part I

Overview

1 | Introduction

Augmenting computational performance by increasing the clock rate of sequential processors has become an unsustainable practice in recent years, due to hardware cost-performance, physical limitations and power consumption requirements. Simultaneously, the computational demands and complexities of modern application domains, such as massively multi-player online games and multimedia data processing, are steadily increasing as consumers call for progressively more complex and intelligent services. To accommodate for this performance demand, contemporary hardware design relies on multiple on-chip processors and specialized hardware to increase performance. This is evident in the emergence of general purpose multi-core CPUs, graphical processing units (GPUs), heterogeneous on-chip architectures, such as the Cell Broadband Engine (CBEA), and networked clusters of these resources. While these new architectures provide considerable opportunities for increasing application performance, the implication of utilizing these resources is that applications need to be written for heterogeneous, parallel or distributed execution environments. This departure from a sequential execution environment imposes considerable overhead, as developing software for parallel or heterogeneous execution environments is significantly more challenging then its sequential counterpart and can greatly complicate the development process.

Due to the complexities introduced by modern hardware design and resource topologies, accommodating for parallel and heterogeneous application development through high-level abstractions, such as programming models, run-time systems, data synchronization techniques and more is essential to ensure that the hardware is correctly and efficiently utilized, while simultaneously ensuring ease of use and adoption by application developers. Formulating and evolving tools and techniques for simplifying the development process is therefore recognized by industry and research communities as a fundamental challenge of contemporary software development. Consequently, there exists efficient and reliable solutions for a wide range of application domains, which facilitate the development process and that limit the impact of the peculiarities of that application domain in a parallel or heterogeneous execution environment. These solutions, however, are not general purpose, and frequently impose artificial limitations that restrict them from being used by other application domains, by focusing on solving a subset of issues specifically related to that application domain, such as data distribution, synchronization, fault tolerance, etc. Providing a general purpose solution that can facilitate the development process of all application domains has proven elusive, as it is hard to predict what requirements future applications might exhibit and the requirements of one application domain might conflict with those of another.

While no general purpose solution for developing applications on modern hardware topologies exists, it is essential to explore the possibilities of creating a unified development process for individual application domains. This process of analysis can provide valuable input into de-

termining what programming models and techniques best facilitate that domain's requirements, with a focus on the simplification of that application domain's development process when utilizing heterogeneous and parallel hardware resources.

Interactive multimedia applications is an emerging application domain and live, interactive multimedia services are steadily growing in volume, with interactively refined video search, dynamic participation in video conferencing systems, user-controlled views in live media transmissions and real-time interaction in distributed virtual environments providing a few examples of features that future consumers will expect when they consume multimedia content. This domain of applications share many characteristics with batch processing applications, a well researched application domain that has broad support for efficient utilization of parallel and heterogeneous execution environments. As such, one would expect that many of the tools and techniques found in these research areas would be applicable to interactive multimedia applications. Interactive multimedia applications have some characteristics that differentiate them in a unique way, however. Interactive multimedia applications have stringent dependencies on time, where knowledge of deadlines is imperative. Additionally, the processing pipelines of these applications typically consists of cycles and complex data dependencies. As such, multiple tools and techniques can provide a basis from which to formulate correct development processes, providing solutions to subsets of problems faced when developing these applications, but where there is a unified approach to supporting the development of a broad range of batch processing workloads no such approach exists for interactive multimedia workloads executing in a parallel or distributed execution environment.

These new usage patterns introduced by interactive multimedia applications, such as extracting features in pictures to identify objects, calculations of 3D depth information from camera arrays, distributed object maintenance, generating free view videos from multiple camera sources in real-time, dynamic migration of state, all add further magnitudes of processing and synchronization requirements to already computationally intensive tasks like traditional video encoding and event processing. Many-core systems, such as GPU's and digital signal processors (DSPs) and large scale distributed systems in general, provide the required processing power, but taking advantage of the parallel and heterogeneous computational capacity of such hardware is much more complex then its sequential counter-part. Therefore, providing tools and techniques to support the development of these applications is imperative, and considering related and comparable work, from research areas such as batch processing, can provide valuable input. Ensuring that hardware topologies and network capabilities remain transparent is essential. The ability to reuse components in different contexts, portability and more, minimize overhead of reimplementing code. Utilizing concepts familiar to the developer to ensure adoption, such as enabling the use of established programming languages. Providing deterministic execution, to simplify debugging and ensure that the correct result is produced for each execution of the application. Finally, efficiency, that the system should be responsive and comparable to hand written, optimized code.

To accomplish these goals, and present the developer with a unified model, considerations from the application characteristics to the data dissemination techniques must be considered as a composed problem. Appropriate programming paradigms or models are required, which can provide a unified abstraction of the hardware resources and provide natural language constructs for the application developer to interact with through a programming language or similar environment.

1.1 Application Domain

Modern computer systems have revolutionized the way we produce and consume multimedia data. This is evident in the emergence of modern interactive multimedia applications, where an immersive environment is created through the combination of multiple content forms, such as text, audio, still images, video or animation. This enables users to connect through a wide range of services, ranging from video conferencing, online gaming, social media, or virtual or augmented reality to name a few. Understanding the requirements of this domain of applications is essential when considering how to support their efficient execution and to ensure that the users have a satisfactory Quality of Experience (QoE).

Intrinsically these services depend on an element of real-time interaction to be functional. This implies that the user's QoE depends on the timely delivery of data or events. If deadlines are consistently violated the service is rendered useless. Deadlines differentiate interactive multimedia applications from application domains that utilize parallel and distributed computing resources to perform computations, where the expectation is merely that the result is provided sooner rather then later, such as batch processing, scientific computing, etc. Having an explicit knowledge of deadlines and being able to act on it is then an imperative feature.

The processing pipeline of an application can be viewed as a graph, where a set of tasks forms the vertices and the flow of data between the tasks represents the edges. For interactive multimedia applications these processing pipelines tend to be quite complex. It is not uncommon for the processing graph to appear near arbitrary in nature, as evidenced by the H.264 video encoder (see section2.2.3), consisting of multiple inter and intra-dependencies between data and tasks. Another prominent feature of these processing pipelines is that they frequently contain cycles, where the output of one stage is used as input to a previous stage, data is continuously refined by some process, such as the k-means workload, or the application receives a continuous stream of data. Due to these complexities, effectively implementing these processing pipelines in code can be quite cumbersome. In addition, these complex patterns can potentially make it a lot harder to determine how and what to parallelize. Having some way of automatically extracting and analyzing these processing pipelines, through a data-flow graph representation or similar, could then provide valuable input.

Due to the real-time and interactive nature of these applications, accurately predicting the behavior of the users is nearly impossible. In the scenario of online games, such as massively multi-player online games, a vast virtual environment is given to roam around in. Due to the immense processing requirements of allowing for thousands of concurrent users to interact in such a manner the virtual environment is typically split into logical regions and distributed to nodes in a cluster or similar to disseminate load across the available computing resources. Given this scenario, it is not possible to predict where in the virtual environment users will be most active and subsequently what server(s) will experience the heaviest load. The unpredictable behavior of the users can result in skewed and seemingly arbitrary performance loads. This unpredictability is not limited to users, but also extends to the application itself, such as encoders and associated transformations of the video data, such as feature recognition. These forms of functionality commonly have variable consumption and production rates, with computational times depending on the amount of content and motion in the video data, resulting in variable execution rates. However, the continuous nature of interactive multimedia applications, and the cyclic nature of the processing pipelines implies that a system can learn from the unpredictable behavior and attempt to extract patterns. This could then enable the application to adapt to the

fluctuations and unbalanced load, which would be highly beneficial.

A processing pipeline, ranging from a massively multi-player online game to a video encoder or multimedia algorithm, is defined as a multimedia workload or merely workload. Another source of uncertainty in modern hardware topologies comes as a result of shared resources; where multiple workloads are potentially competing for the same resources at any given time. This is becoming increasingly common as a result of the extensive use of hypervisors and virtual machines, exemplified by modern cloud computing services, such as Amazon's EC3. In these services the physical resources of a computational node are divided into logical slices and divided among multiple customers. Being able to adapt to changes in available hardware resources is therefore a primary concern as well.

1.2 Heterogeneous, distributed hardware resources

The evolutionary path of interactive multimedia applications is closely linked to the developments occurring within the hardware industry. The processing requirements of modern multimedia workloads are immense, such as rendering a 3D-overlay in an augmented reality application. As a result, the performance of these applications is commonly only limited by the available hardware resources.

Software developers have consistently pushed the boundaries for computational demand. As the physical limitations of pushing the clock frequency on a single- core CPU became apparent, new hardware solutions evolved. The focus moved towards heterogeneous support chips, then parallel processing clusters and transputers followed by symmetric multi-chip processing and finally multi-core processing and general purpose graphical processing units. This effectively forced the programmers to parallelize their applications to increase performance.

The variations in hardware architectures also impact how distribution and synchronization of state is performed, where shared and non-shared memory architectures require fundamentally different approaches. Parallel applications on shared-memory architectures require synchronization of threads of execution when accessing shared data. Using synchronization primitives correctly requires extensive experience, otherwise one might thrash locks, serialize parallel steps, or create deadlocks, starvation, race conditions, etc. On non-shared memory architectures, mechanisms such as message passing or active messages must be utilized. Furthermore, non-uniform memory access on a single node computer also requires careful consideration as to the placement of data. Moving from a sequential mode of operation to a parallel execution environment is significantly more demanding on a developer.

Today, most commodity programs can run on a multi-core machine, but there is a number of applications, such as real-time high definition video encoding or decoding and 3D rendering, typical components in interactive media application pipelines, that still require offloading to specialized hardware. Heterogeneous resources, such as modern GPUs provide some of the hardware resources to accomplish these tasks efficiently.

The emergence of the GPU has led to general purpose graphical processing unit programming (GPGPU), evident in the release of the OpenCL and CUDA frameworks. These frameworks provide APIs and compilers that attempt to conceal the fact that tasks will be executed on a GPU, but knowledge of structuring programs optimally is still required. Specialized heterogeneous resources can therefore increase performance, but require that the developer obtains knowledge of how to effectively utilize the hardware.

While a number of applications benefit from the use of specialized hardware, many remain computationally bound even then. Consider a massively multi-player online game (MMOG) with thousands of concurrent users interacting in a virtual environment. For this type of application it is necessary to move beyond single node resources. Distributed computing, which coordinates the processing capabilities of multiple machines, can make available resources beyond the capabilities of single node machines. By using these resources it is then possible to accomplish tasks in significantly reduced time. Though this approach is not without problems, communicating via a network adds latency and can introduce delays detrimental to the progress of the application. It also becomes necessary to distribute data and synchronize state between nodes in the topology. This last point is also prominent in single-node architectures if a GPU is used to offload computations to. Fault-tolerance is required to handle situations where nodes in the network become unavailable, and these are just a few of the obstacles that must be overcome. As such, distributed computing improves the scalability of an application, but introduces overhead (communication primitives, scheduling, synchronization, fault-tolerance, etc.) and the developer must consider this when partitioning the work.

Developing applications that utilize distributed heterogeneous computing resources is a hard task. Ensuring that the utilization of the resources is maximized is even harder. This means that it is essential to develop a framework that makes it possible for a programmer to write an application in a manner close to their intrinsic understanding of how to structure a program, while simultaneously ensuring that the run-time system executing the application is capable of maximizing the utilization of the available resources.

1.3 Range of parallel programming solutions

While executing tasks in parallel can lead to a considerable performance gain, implementing an application that efficiently utilizes the available resources is a very demanding process. As a result, a number of solutions provide support at various levels, either by providing methods for annotating code to achieve automatic parallelization or through application design, with novel languages or entire execution engines or run-time systems.

Proposed solutions range from parallelizing compilers (Intel TBB [1], Fortran D [2], Paradigm [3], Polaris [4]), parallel languages (Sisal [5], Data Parallel Haskell [6], StreamIT [7]), message passing (PVM [8], MPI [9]), virtual shared memory (Linda [10]), distributed shared memory (Ivy [11], Mirage [12]) to execution engines (MapReduce [13], Dryad [14]).

A number of these solutions are designed to solve specialized tasks, such as distributed communication or parallelizing loops, or provide support for a particular domain of problems, such as batch processing or large scale numerical computations. They all have in common that their usefulness in one application domain can limit them in another. Parallelizing compilers typically only work on shared memory architectures, as the latencies introduced by network communication make it hard to create proper optimizations. Parallel languages require learning a new language, and typically function well for subsets of problems, such as numerical calculations or data parallelism or they extend a functional language with parallelization, which is an abstraction that can limit the ability to easily express a number of workloads. Distributed shared memory still leaves the tasks of function and data partitioning, as well as locking of shared resources to the developer. The execution engines are typically designed to solve batch processing workloads, and consequently have limitations in their processing pipelines, such as

only supporting directed flows of data.

The range of proposed solutions is broad, and exhaustively determining their applicability to our target application domain, of interactive multimedia applications, is therefore not feasible. However, reducing these approaches to their concepts and evaluating the concepts applicability to our application domain is possible. These approaches can therefore provide a valuable starting point from which to eliminate or add concepts and paradigms that are suitable for supporting our domain of applications. The identification of these concepts is not explicitly addressed in our submitted articles due to limitations in space and their insufficient individual contributions. The body of work that comes as a result of our iterative refinement of these concepts, through extensive prototyping and the continual inclusion of workloads, presents a significant contribution however, and is summarized in chapter 3.

1.4 Motivation

A lot of research has been dedicated to addressing the challenges introduced by parallel and distributed programming. This has led to the development of a number of tools, programming languages, run-time systems and more to ease the development effort.

The processing and development of distributed multimedia applications is inherently more difficult than traditional sequential batch applications due to the number of concepts that must be utilized simultaneously to achieve a result. Multimedia applications have strict requirements and knowledge of deadlines is necessary, especially in a live scenario. For multimedia applications that enable live communication, iterative processing is essential. Also, elastic scaling with the available resources becomes imperative when the workload, requirements or machine resources change.

To expect an application developer to be able to have complete knowledge of how to best utilize all possible hardware topologies is unrealistic. Furthermore, statically designing an application to run on the available resources and correctly implementing a distributed application capable of dynamically adapting to the workloads and resources is a cumbersome process.

The motivation for this thesis is to evaluate programming models, run-time systems and communication primitives and extract the fundamental concepts that compose them to evaluate their applicability to interactive multimedia workloads. The main body of work in this context is not a result of the extraction of these concepts, though this does represent a significant contribution, but rather the successful combination and implementation of these.

The expectation is therefore to be able to harness the knowledge gained from this process and apply it to the formulation of a high-level programming model and subsequent run-time system.

1.5 Research objectives

Interactive multimedia applications are an emerging domain of applications. The unique characteristics of the workloads composing these applications, exemplified by interactivity, complex interactions between tasks and data and arbitrary processing graphs means that implementing these applications is highly demanding. This fact is further complicated by their high demand for computational power, which in the existing hardware landscape means utilizing multiple cores, heterogeneous hardware or distributed systems. Efficient utilization of these resources is

a complicating factor, where existing models, tools and techniques have limited applications to this domain of applications. The purpose of this thesis is therefore to increase the knowledge surrounding interactive multimedia applications and their unique characteristics, and how these characteristics could best be supported by parallel programming models, run-time systems and related tasks. This thesis thereby contributes to a larger research topic, where the overall goal is to support the development process and execution of interactive multimedia applications; by reducing the cognitive load on the developer through the formulation of a high-level programming model and the subsequent implementation of run-time systems that can efficiently utilize heterogeneous and distributed hardware resources.

To this effect, a number of contributions have been made to specific research objectives (RO) to further the understanding of how these parallel programming models, run-time systems and communication models interact with the application domain of interactive multimedia applications.

RO1 *Workloads:* Determine what parallelization opportunities exist in interactive multimedia application workloads. Extract the unique characteristics of these workloads and determine their potential impact on subsequent investigations.

RO2 *Abstractions:* Evaluate existing parallel programming models and extract concepts that are relevant to the criteria defined in RO1 and formulate a composition of concepts on this basis.

RO3 *Execution:* Explore the feasibility of supporting the execution of interactive multimedia workloads given the definition of a workload in a high-level parallel programming model as formulated by RO2.

RO4 *Validation:* Validate the concepts extracted from RO2 and RO3 and their applicability to the requirements of RO1 through prototyping.

1.6 Scope and delimitation

This thesis has focused on the formulation of a new parallel programming model applied to the emerging application domain of interactive multimedia applications; based on an evaluation of existing models, tools and techniques. With the formulation of this new high-level parallel programming model, efficient run-time system support for the execution of these workloads is expected. A long-term goal is to eventually achieve optimal scheduling of tasks composing a workload through novel scheduling algorithms.

A primary objective was therefore to provide the ability to formulate a programming model that makes it possible to extract run-time information about workloads and organize this in a way that makes it easy to analyze, using a graph representation. The type of information we are interested in extracting from the execution of a workload is time used for dispatching, executing and waiting, number of messages sent and their sizes and cache hits and misses. The intent of this thesis was to lay the ground work for future work into the leveraging of this information to create optimal or near optimal scheduling algorithms. Furthermore, this information could be made persistent and use to statically analyze the workload retrospectively and perform compile-time optimizations in addition to run-time optimizations.

While these forms of run-time and compile-time optimizations were a long-term goal, they were beyond the scope of this thesis to thoroughly evaluate. The contribution of this thesis has therefore been to provide important ground work for the future exploration of these research areas. The focus of this thesis has therefore been on the formulation of a high-level programming model and the subsequent run-time system implementation for executing workloads as a proof-of-concept. We have not developed or evaluated new scheduling algorithms, approaches to dividing the workload, performing static analysis of the workloads or utilizing distributed topologies of resources within the context of this thesis.

Furthermore, we have made the assumption within this thesis that all code is shared between the interacting computing nodes. We are aware of the problems related to distributing code in large clusters of compute nodes, but have not considered this a fundamental part of our system.

1.7 Research method

To arrive at our conclusions we have used prototyping extensively, because prototyping has the benefit of providing early feedback as to the applicability of a given concept to a problem domain. At a relatively early stage in the development process it is possible to determine whether the approach matches the requirements of the project. Through prototyping one can continuously evaluate whether the interaction between the developer and programming model matches the users requirements.

In this thesis we have made use of both rapid and evolutionary prototyping. Rapid prototyping refers to the creation of a model that will eventually be discarded, but can provide valuable input to the incremental formulation of new prototypes. This has enabled us to evaluate alternative approaches and determine their applicability to the problem domain. Evolutionary prototyping allowed us to discover limitations in our approach and rectify them through subsequent iterations. The purpose of evolutionary prototyping is to build a very robust prototype in a structured manner and constantly refine it, where the prototype can eventually become an integral part of a new system; a long-term goal of this thesis.

1.8 Main contributions

With a firm basis in rapid and evolutionary prototyping, as discussed in section 1.7 and contributing directly to RO4, we have pursued solutions to solving the issues raised by the research objectives outlined in section 1.5, through the evaluation of workloads and multiple prototypes.

Our contributions have been evolutionary in their approach, where the rapid prototypes of Ginnungagap [15–19] and Nornir [20–22] provide input to their respective research objectives. The formulation of workload requirements and parallelization opportunities as summarized in chapter 2 contribute to RO1, parallel programming model considerations, as summarized in chapter 3 contribute to RO2 and run-time system support and communication models, as summarized in chapter 4 and 5 contribute to RO3. The subsequent evaluation of these contributions have led to the formulation of P2G [23–25], which provides a unified view of research objectives RO1 through RO3.

The main contributions presented in this thesis have been published in a number of research papers. In the following, we look more closely at the specific contributions each prototype has had to the individual research objectives.

1.8.1 Deferred decisions

Providing the ability to defer important decisions to run-time is essential, making it possible to perform load-balancing and optimize scheduling and task execution. The concept of deferred decisions was first evaluated with Ginnungagap, where predicting the load of a given server or the interaction patterns of users is fundamentally impossible to achieve during deployment of the application. Being able to dynamically relocate users or change the granularity of the environment to reduce load on demand could then increase the scalability of the solution. With Nornir and P2G, being able to determine during run-time where a task should be executed or how a workload should be partitioned would also aid in the adaption to the available resources or characteristics of the workload. Essentially, interactive multimedia applications are intrinsically unpredictable, being able to defer essential decisions to the run-time execution of a workload is very important, as summarized in chapter 4.

1.8.2 Decoupling

In Ginnungagap we evaluated functionality for decoupling the object model the developer interacted with from the run-time system. This was achieved by extending the syntax of the programming language with keywords that would ensure functionality for migration and remote method invocations. This was a relatively simple implementation, but greatly facilitated the development process by generating code that the application programmer interface was used correctly by ensuring that state was serialized correctly before being transmitted and ensured that method invocations were evaluated before being invoked locally or redirected to a remote process or node.

In Nornir, this approach was not utilized and as a result defining the interaction patterns between tasks and data proved to be a cumbersome and error-prone task. An essential part of P2G was therefore to determine how decoupling of the high-level programming language from the run-time system could be achieved in an efficient way. A primary contribution of P2G is the identification of an efficient way to allow for parsing and code-generation of the high-level programming language while utilizing the benefits of an existing compiler. This was achieved by implementing a thin-layer called a coordination language, which defined a syntax for tasks and data interaction, as summarized in section 3.4. The developer then implemented their tasks using C++ and specified the interaction with data using the coordination language. The parser ensured that the semantics of the programming model were guaranteed and generated code that integrated with the run-time system on this basis.

1.8.3 Distributed name service

A distributed name service supports the efficient maintenance of references to objects in a topology of processes or nodes in a distributed system, as summarized in section 5.2. This form of reference maintenance has previously been applied to distributed shared memory for maintaining references to pages of memory in distributed systems with complex interconnection network topologies. With Ginnungagap, we extensively evaluate the feasibility of applying this technique to object reference maintenance in a wide-area network, where an object in this case is an instance of a class as per the object-oriented paradigm. A contribution of this thesis is thereby to the successful evaluation and application of this technique to provide reference maintenance of objects.

While applying this technique to a distributed version of P2G was beyond the scope of this thesis, P2G was designed to use this form of reference maintenance as a result of the evaluations made in Ginnungagap.

1.8.4 Migration or resource locality

A name service is closely related to migration, where the name service makes it possible to migrate or replicate state and maintain references to the state as it moves between the nodes in the topology. In distributed shared memory (DSM), migration is used to transfer the ownership of a write-page, i.e., a page that can only be written to by one party at the time, or to create replicas of pages. Maintaining references in this context is important because writing to a write-page can invalidate the read-copies. Migration in DSM is used to ensure resource locality.

Being able to migrate state between processes or nodes in a distributed system of resources to ensure resource locality can be vital to ensure that a sufficient Quality of Experience (QoE) is achieved, because optimizing the location of state relative to the interacting parties is then possible. Providing the efficient migration of state is a major contribution of this thesis through the work done in Ginnungagap, as summarized in section 5.3.1.

1.8.5 Explicit and implicit dependency graphs

Due to the complex data and task inter-dependencies of modern interactive multimedia workloads, as exemplified by the H.264/AVC workloads defined in section 2.2.3, finding a suitable abstraction for specifying the interaction between communication and computation is necessary.

Explicit dependency graphs, as evaluated in Nornir, then proved to be an intuitive representation of the interaction between tasks and data in a workload, where the vertices represent the computation or tasks and the edges represent the communication channels between the tasks. With Nornir we gained an understanding of the usefulness of this representation through the formulation of explicit dependency graphs, i.e., the developer was responsible for manually specifying the communication channels between tasks. However, efficiently formulating these explicit communication channels and their subsequent graph representation proved cumbersome, time consuming and error-prone.

In P2G we further evaluated this approach by looking at the possibilities of implicitly defining the relationships between tasks and data. This then gave us the possibility of harnessing the power of explicit communication, but arriving at it implicitly through a high-level abstraction. The contributions to the dependency graph model are summarized in section3.3.5.

1.8.6 Compositionality

Compositionality is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. Interactive multimedia applications are typically compositions of multiple tasks, where these tasks can form logical components that can be constructed to create a larger application. From these smaller building blocks, using compositionality one can then deterministically guarantee that the composed application will produce the same results given the same input. A major contribution of Nornir and subsequently of P2G was to utilize this approach, as summarized in section 3.3.5.

1.8.7 Determinism

With Nornir and P2G we evaluated the benefits of deterministic execution in concurrent systems, as summarized in section 3.3.2. In essence, a deterministic computation will always produce the same output given the same input. A non-deterministic execution of tasks can not guarantee that the same output will be produced, which can, in the worst case, lead to subtle differences in the output.

In Nornir, deterministic execution was achieved through unidirectional point-to-point communication channels, where polling was not allowed as a result of the formal semantics of Kahn Process Networks. While this guarantees the deterministic execution of tasks, the channel abstraction proves cumbersome for interactive multimedia applications, as they typically operate on subsets of named state.

To ensure deterministic execution in P2G, we explored possibilities of maintaining these semantics while operating on named state.

1.8.8 Named state

Named state refers to shared and accessible state, while unnamed state implies that there is no way of directly accessing the current state of the application. For a number of interactive multimedia workloads interaction with named state forms a typical part of the algorithm. For workloads like H.264/AVC and *k*-means this makes sense, because there are a number of tasks that can be executed simultaneously on the same subset of data. Passing this data through explicit channels, such as one would in Nornir, is possible, but due to the complex interaction patterns, reading and writing from a shared data structure that is reachable from all tasks makes it easier to define these interactions. The use of named state coupled with deterministic execution of tasks is a primary contribution of P2G.

1.9 Disposition of the thesis

We have structured this thesis into two parts. In **Part I**, we provide a context and overview of the thesis work, while **Part II** contains the individual research papers.

The remainder of **Part I** is structured as follows. In chapter 2, we discuss parallelization opportunities in interactive multimedia applications, where we provide some context for the work carried out in this thesis. In chapter 3, we examine the concepts that form the abstract definition of a parallel programming model in light of the workload requirements described in the previous chapter. In chapter 4, we review the role of run-time systems in modern computer application design. Then in chapter 5 we look at the communication models that allow for the dissemination of state between processes and nodes in a topology of hardware resources. In chapter 6 we review the knowledge gained in the previous chapters in light of the papers we have published as a result of our research. Finally, in chapter 7 we summarize the main findings of our thesis and evaluate to what extent we have achieved our research objectives.

2 | Parallelization opportunities in interactive multimedia applications

With the emergence of new hardware paradigms, parallelizing workloads to increase application performance has become increasingly common. The imperative programming paradigm, which is the most prominent programming paradigm, induces complex cognitive models as a result of non-determinism, which occurs naturally as a result of concurrent execution of sequential threads. An application developer might therefore spend considerable time on minimizing or removing the effects of this non-determinism. This is unfortunate, as it is a tedious, complex and error-prone task, as wielding low-level locking mechanisms is a non-trivial exercise. Considerable research has been dedicated to reducing the cognitive overhead of parallelization through auxiliary programming models, run-time systems, novel synchronization mechanisms and more. However, this proliferation in support for parallel application development only extends implicitly to the domain of interactive multimedia applications as inadvertent inter-domain convergence. The implication being that explicit support for parallelizing workloads, or components within an interactive multimedia application is limited.

A primary focus of this thesis is to provide system support and cognitive models that can alleviate interactive multimedia application development in parallel execution environments. Multimedia workloads introduce limitations that other workloads do not need to care about, i.e., deadlines and bandwidth consumption. Intrinsically multimedia applications depend on an element of real-time interaction to be functional. This implies that the user's Quality of Experience (QoE) depends on the timely delivery of data or events. If deadlines are consistently violated the service is rendered useless. Furthermore, due to these time constraints and the bandwidth consumption of high-definition video, it is critical that the system can deliver. Where both system support and the formulation of a cognitive model, which is capable of providing appropriate abstractions, requires domain knowledge and a thorough understanding of the characteristics and requirements of the applications. Furthermore, it is necessary to find a balance between the granularity of the high-level abstractions and potential performance penalties they might impose. Ideally, the developer should be able to model the application using intuitive language constructs, simultaneously as we ensure that the inherent parallelism of the application can be captured at a reasonable level of granularity, to ensure full utilization of the hardware resources that are available.

The purpose of this chapter is therefore to provide some insight into the parallelization opportunities that exist within the domain of interactive multimedia applications for the purpose of more readily extracting the requirements this will place on how to evolve methodologies and run-time systems. We will accomplish this by looking at components that commonly comprise these application and have elected to focus on the H.264/AVC video coding standard [26], the

k-means clustering algorithm [27] and event processing. Our past experience has taught us that these workloads encompass general traits and fundamental algorithms that represent the specter of obstacles one can encounter when developing interactive multimedia applications for parallel execution environments. In the following sections we will discuss program structure in relation to the workloads we have selected for further examination.

2.1 Decomposition domains

We can create a decomposition tree, as seen in the lower part of figure 2.1, by separating an application into its constituent elements or parts. Evaluating such a tree then makes it easier to evaluate the application domain in question.

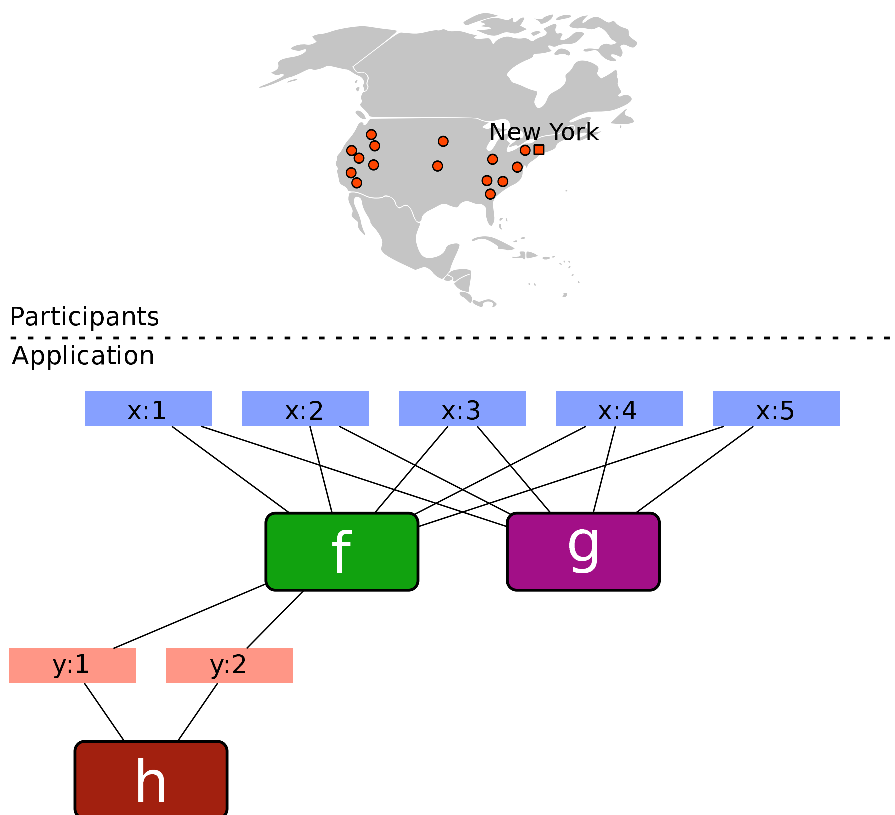


Figure 2.1: Decomposition tree

In the case of figure 2.1, we can see that we are viewing the application in relation to the participants that are interacting with it. This makes sense, because, as we have seen with deadlines in interactive multimedia applications, the requirements of the participants can have a profound impact on the application design itself.

The decomposition tree itself is the result of evaluating the individual components of the application and their interaction. The types of decomposition that we most frequently discuss are those of functional, data, nested and partitioning.

Within this context, functional decomposition refers to the act of dividing code into distinct components, which is shown as the boxes labeled *f, g, h*. During sequential execution these components are executed according to a predetermined call graph (excluding the scenario where random variables can impact the call graph). Nonetheless, for sequential scenario the entire execution of the application is restricted to single core resources. However, if parallel execution of these components is possible, we can greatly increase the execution time by concurrently executing components that have their dependencies met. As such, when the data in the array x is made available, the implication of functional decomposition is that f and g can run in parallel.

If either f or g operate on individual data points in x we are moving into the domain of data decomposition. We define data decomposition as the ability to subdivide a set of data into distinct elements. Data decomposition is useful in scenarios where some unique piece of code, such as a procedure or function, is applied successively to each element in the data set, such as f being applied successively to *x:[1-5]*. In this thesis we are primarily interested in a more narrow definition, where this unique piece of code also can be applied to each element in an arbitrary order (as long as any dependencies to execute the code on that data have been met). This gives the implied side-effect that we can process these elements concurrently or in parallel. This ability to apply code concurrently to each element of a decomposed data structure, rather than successively, can significantly speed-up an application.

By looking at function and data decomposition as a composed problem, we can see that nested decomposition evolves naturally. Nested decomposition refers to the fact that we can decompose at multiple levels within an application. For example, the function f, when operating on a data point in x, internally partitions the data further by applying the function h to *y:[1-2]*. As such, we have to consider what the hierarchy of function and data decomposition looks like.

2.1.1 Partitioning

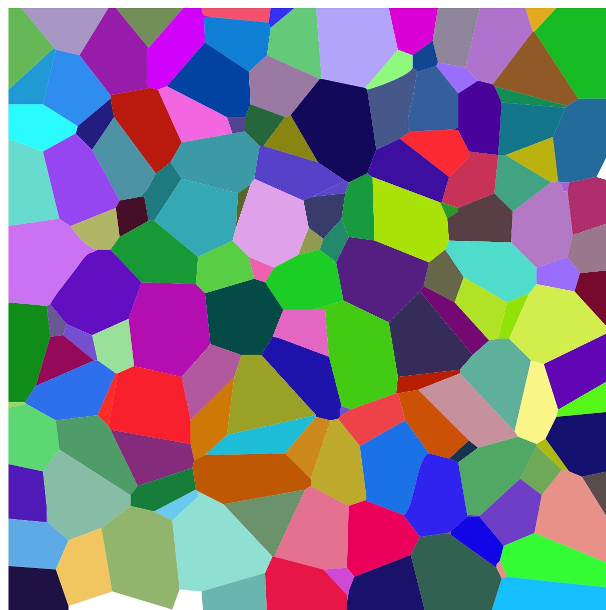


Figure 2.2: Colored Voronoi 3D slice (from http://en.wikipedia.org/wiki/Voronoi_diagram)

Partitioning is possible only by virtue of the application itself, i.e., when the application

itself models a decomposable system. Under such circumstances, the interacting parties and their related state within the application can be divided into disjunct sets, much like a Voronoi diagram (see figure 2.2), where each set can be viewed as a Voronoi cell, where these cells represent an *area of interest*.

What we have described here is a very coarse grained form of decomposition and typically the application code, in its entirety, is replicated at each distinct location. The disjunct sets, or areas of interest, are then processed or handled individually, as such, the scope of events that need to be processed can potentially be greatly limited.

It is not implied that such an architecture limits the flow of information to within the set or that the interested parties of a set are static. If we follow the Voronoi diagram analogy, we can imagine that where the cell walls meet between cells there might be a necessity to exchange state. Furthermore, it is fully possible that an interested party can change their subscription from one set to another during execution or even be subscribed to multiple sets simultaneously.

2.2 Workloads

We will now introduce the workloads we have selected for further discussion. During the remainder of this chapter we will discuss internal details of these workloads, with a focus on decomposition and their subsequent impact on parallelization opportunities.

2.2.1 Event processing

One of the distinguishing features of interactive multimedia applications is their strict timing requirements, i.e., the user experience is dependent on the timely arrival of events or data. For interactive multimedia applications with physically distributed participants, operating in a distributed system of resources, this timeliness requirement is particularly important.

This is important, because under the circumstances just described, network traversal can add a dimension of latency to an already time sensitive application. Furthermore, these applications are typically already time restricted due to computationally intensive workloads.

Consider an MMOG that is built to support thousands or even tens of thousands of concurrently interacting users, with each user generating a number of key events every second, i.e., events that cannot be dropped without causing serious side-effects. For such a system to receive, validate the content of the event and generate corresponding results would take considerable resources to achieve, particularly within the time-constraints given by these applications.

Utilizing distributed systems can therefore be helpful in a number of ways, by negating resource limitations, because of an increase in the number of hardware resources available, and a set of distributed proxies can make it possible to reduce the latencies of the interacting parties, by reducing the physical distance to a primary server (as seen in figure 2.3). However, utilizing these resources efficiently is still difficult and requires that the application to some extent supports such an execution environment. Nonetheless, efficient event processing when operating in a distributed system is vital. Particularly for tasks such as evaluation, distribution and dispatching of events.

As described, the application itself needs to support the possibility of utilizing these distributed resources. To this effect, massively multi-player online games (MMOGs), are a salient example of a class of interactive multimedia applications that are inherently decomposable at

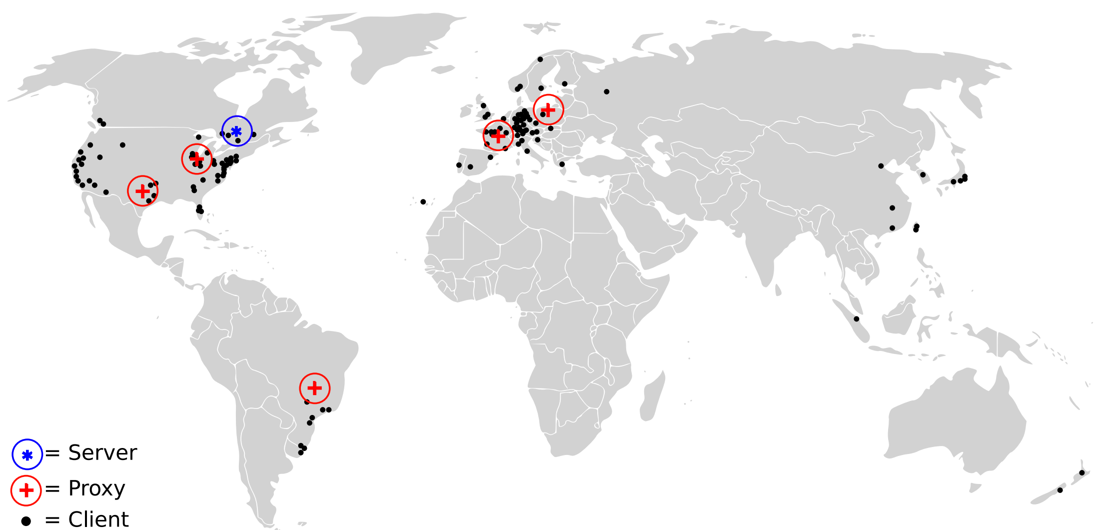


Figure 2.3: Location of players relative to servers

application level, i.e., it is possible to partition the application based on its participants and their interaction patterns (as discussed in section 2.1).

This is the case, because MMOGs create a virtual environment (as seen in figure 2.4), which simulate real or imagined environments, such as medieval landscapes, alien worlds, etc. These virtual environments are then populated by users' avatars, i.e., virtual representations of themselves. As MMOGs simulate the real world, we can use proximity between entities to create boundaries, i.e., as in the real world we only observe those people who are in a reasonable proximity to ourselves. Therefore MMOGs are an example of applications that can be logically decomposed, i.e., we can divide the virtual environment into logical regions, and thus distribute the interaction of the users to different nodes in a distributed system.

As we describe in [15–17], dividing the virtual environment into these logical regions is a commonly applied strategy to ensure scalability. Partitioning of the application then makes it possible to solve deadlines within a partition, and the only influence deadlines have on the application are how many partitions you need.

When virtual regions are created the boundaries can either be explicit or seamless, where a seamless boundary gives the best user-experience. However, if these boundaries appear to be seamless, this also means that there needs to be some form of synchronization between the regions themselves.

Furthermore, users are mobile, and can move between regions, as such, there must be mechanisms in place to migrate users between these regions.

In conclusion, partitioning can be a powerful tool to provide scalability, in terms of processing power and participants in an application. There is, however, no guarantee that the partitions are mutually exclusive, with regards to data, frequently some synchronization is required.

2.2.2 *k*-means clustering algorithm

k-means clustering is an iterative algorithm for cluster analysis, which aims to partition n data points placed within a d dimensional space into k clusters, where each data point belongs to the cluster with the nearest mean. This algorithm has many applications and is, among other

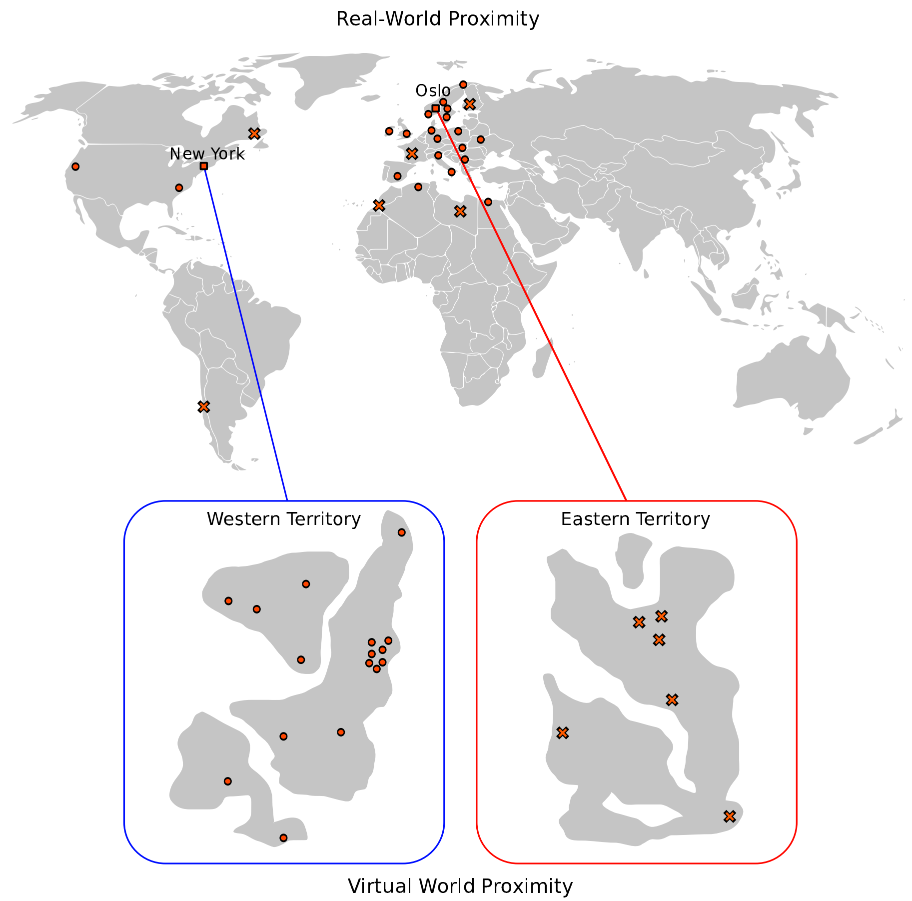
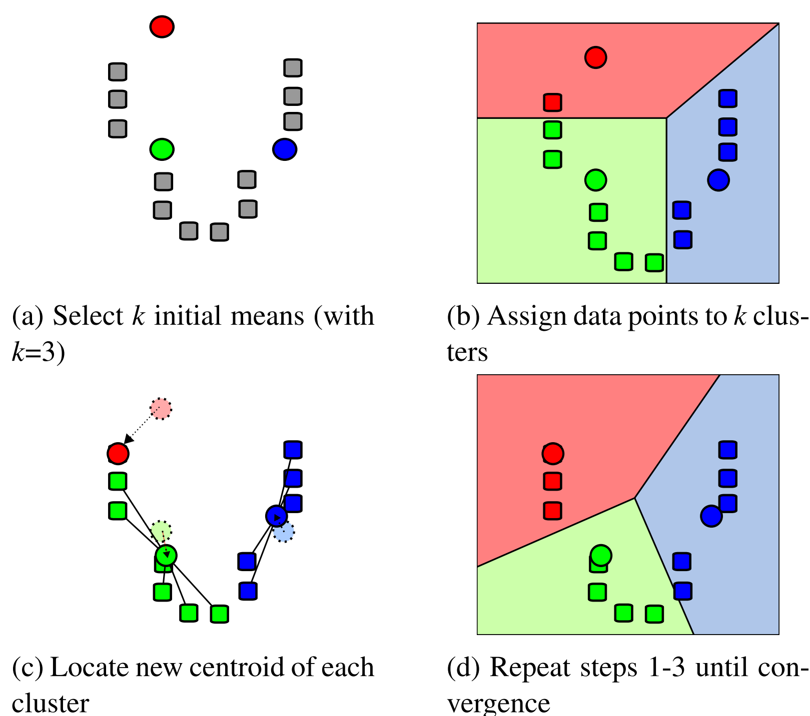


Figure 2.4: Partitioning

things, frequently used as a preprocessing step for other algorithms. In feature recognition, for example, k-means is used to identify objects (or features) found within an image that then can be extracted and compared to a database of known features. The clustering itself can also be based on a number of different parameters, such as calculating the euclidean distance between a point and the cluster centroids, or measuring the difference in intensity or color. k-means is generally considered an NP-hard problem for values of d larger than 2. Thus, there exists a variety of heuristic algorithms, though in this thesis we will present the basic k-means algorithm.



(a) Select k initial means (with k=3)

(b) Assign data points to k clusters

(c) Locate new centroid of each cluster

(d) Repeat steps 1-3 until convergence

Figure 2.5: k-means clustering algorithm. Source of image *Wikipedia*, http://en.wikipedia.org/wiki/K-means_clustering.

Workload description

In figure 2.5, we can see how the k-means algorithm unfolds for k=3 and d=2. Initially (see figure *2.5a*), k data points are randomly selected as the centroids of the clusters. There are other strategies to selecting the initial centroids also, but for the purpose of this explanation this is the most straightforward approach. Next (see figure *2.5b*), we assign each data point to a cluster of the nearest mean, which is located by calculating the euclidean distance to the centroid of each cluster and selecting the closest one. Then we calculate the new centroid of the cluster by calculating the mean of the cluster (see figure *2.5c*). These steps are then repeated until we reach convergence, i.e., the centroids are the same between iterations, or we have reached a predetermined number of iterations.

As mentioned initially, this algorithm has a number of uses within multimedia workloads. For instance, in real-time processing it is frequently used in color image segmentation, which can be useful in time sensitive scenarios, such as when sorting items on a conveyor belt based on color. Furthermore, in such a scenario, it is vital that the algorithm converges as fast as possible, otherwise the sorting might fail, due to too many false positives, or be to slow to be able to handle the load it is given.

Decomposition

It is therefore important that we are able to support the efficient execution of these types of workloads. To achieve this we need to understand something about the program structure of the workload. Looking at figure 2.6, we can see how the algorithm, as described in figure 2.5, can be decomposed. Discounting the selection of the initial centroids, the first stage of the algorithm is to assign each data point to a cluster, based on a distance function. As we can

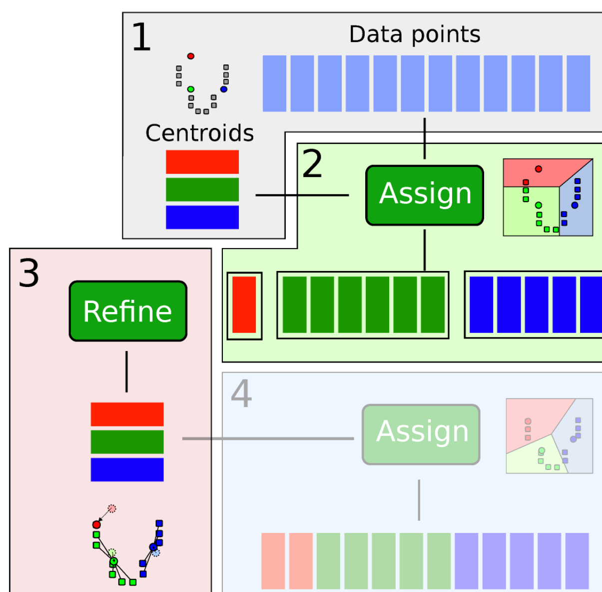


Figure 2.6: k-means decomposition tree

see from the decomposition tree, the assign function relies on data from two data structures. Furthermore, there are a total of $d * k$ operations that can be performed at this stage, without any strict ordering requirements. At this point there is a barrier between the *assign* and *refine* stage, i.e., we are not able to calculate the new centroids of the clusters completely before all data points have been assigned a cluster. Once the data is available though, we can calculate the k new centroids, one for each cluster. As such, using the decomposition method of evaluating a workload, we can more readily make informed choices when determining how to support the parallel execution of such workloads.

2.2.3 H.264/AVC

Finally, we will look at the H.264/AVC video coding standard [28], which is a joint venture, formalized as *Recommendation H.264* by the Video Coding Experts Group (VCEG) and as *International Standard 14 496-10 (MPEG-4 Part 10) Advanced Video Coding (AVC)* by the ISO/IEC Moving Pictures Expert Group (MPEG). Coinciding with the stated goal of these standards, the coding efficiency in H.264 is vastly improved compared to its predecessors MPEG-4 Part 2, H.263, etc. [26] and reduces the required bit-rate to encode a video at a given quality level by up to 50 percent in most cases. H.264 can therefore deliver a higher resolution or frame rate at lower bandwidth with the same quality level as previous standards, which makes it ideal for delivering high definition (HD) services over bandwidth restricted transmission mediums. Coupled with an improved focus on support for greater variety of communication networks, H.264 is also ideal for delivering content to mobile devices on unreliable networks, such as smart phones, laptops or other devices operating on wireless networks or networks with high packet loss, latency, etc.

Like its predecessors, H.264 utilizes a block-based hybrid video coding approach and rather than approach video encoding in a new way, rather improves significantly on existing methods for reducing redundancy in the spatial, temporal and statistical domains. This is achieved through the introduction of variable block-size, subpel motion compensation, integer transform

Table 2.1: H.264 Profiles

	Baseline	Extended	Main	High	High 10	High 4:2:2	High 4:4:4Predictive
I and P Slices	Yes	Yes	Yes	Yes	Yes	Yes	Yes
B Slices	No	Yes	Yes	Yes	Yes	Yes	Yes
SI and SP Slices	No	Yes	No	No	No	No	No
Multiple Reference Frames	Yes	Yes	Yes	Yes	Yes	Yes	Yes
In-Loop De-blocking Filter	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CAVLC Entropy Coding	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CABAC Entropy Coding	No	No	Yes	Yes	Yes	Yes	Yes
Flexible Macroblock Ordering (FMO)	Yes	Yes	No	No	No	No	No
Arbitrary Slice Ordering (ASO)	Yes	Yes	No	No	No	No	No
Redundant Slices (RS)	Yes	Yes	No	No	No	No	No
Data Partitioning	No	Yes	No	No	No	No	No
Interlaced Coding (PicAFF, MBAFF)	No	Yes	Yes	Yes	Yes	Yes	Yes
4:2:0 Chroma Format	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Monochrome Video Format (4:0:0)	No	No	No	Yes	Yes	Yes	Yes
4:2:2 Chroma Format	No	No	No	No	No	Yes	Yes
4:4:4 Chroma Format	No	No	No	No	No	No	Yes
8 Bit Sample Depth	Yes	Yes	Yes	Yes	Yes	Yes	Yes
9 and 10 Bit Sample Depth	No	No	No	No	Yes	Yes	Yes
11 to 14 Bit Sample Depth	No	No	No	No	No	No	Yes
8x8 vs. 4x4 Transform Adaptivity	No	No	No	Yes	Yes	Yes	Yes
Quantization Scaling Matrices	No	No	No	Yes	Yes	Yes	Yes
Separate Cb and Cr QP control	No	No	No	Yes	Yes	Yes	Yes
Separate Color Plane Coding	No	No	No	No	No	No	Yes
Predictive Lossless Coding	No	No	No	No	No	No	Yes

and improved entropy encoders. Introducing these new and improved measures comes at a cost, however, as they cause a considerable increase in implementation complexity and require considerably more processing power during encoding. Due to this increase in implementation complexity and in performance requirements, the incentive for supporting the development of these types of applications in a parallel execution environment is therefore considerable. The potential speed-up provided by a multi-core architecture is considerable, as such, if it is possible to negate the cognitive penalty of switching to such an architecture the choice is obvious.

As has commonly been the case for ISO/IEC video standards, only the central decoder is standardized. This guarantees that every standard conforming decoder will produce the same output given the same input. To facilitate the possibility of having a mismatch between encoder and decoder capabilities, a number of profiles and levels have been defined by the H.264 standard. Then the implementation can easily signal what profile(s) it supports, see table 2.1 for an overview of the profiles. The implication of these facts is that there is considerable flexibility as to how an encoder can be implemented, both in terms of its support for different H.264 profiles, but also in terms of encoding quality. This can be beneficial as an encoder implementation then can be tailored to operate optimally for the environment it runs in, such as a parallel execution environment.

Of the workloads we have selected, H.264 is by far the most complex and comprehensive one, and can be viewed as a composition of a number of smaller workloads, as we will see in the next section.

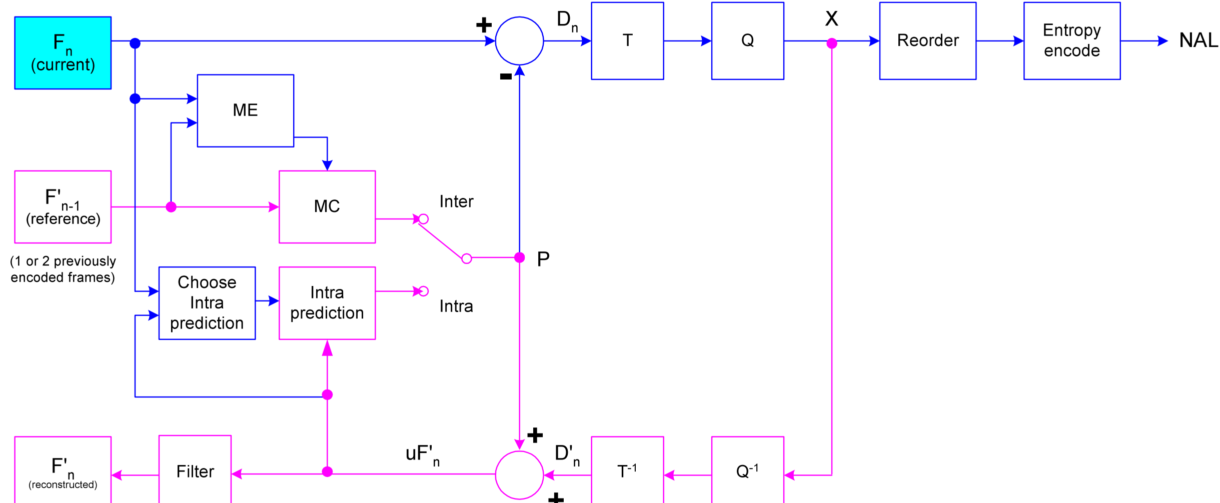


Figure 2.7: H.264/AVC Encoder, figure by I. Richardson [29]

H.264/AVC Encoder

In figure 2.7, we provide an overview of the main sequence of events during the encoding process of a valid H.264/AVC bit stream.

In essence, the purpose of an encoder is to take single shots of unprocessed image data, called frames, and compress this data into a smaller size with a minimal amount of loss in quality. This is achieved by exploiting spatial and temporal redundancy within, respectively, the current frame or previously encoded frames.

While this may sound relatively simple, figure 2.7 gives us some insight into the complexity of this process. The encoding process starts by analyzing the current frame F_n. This analysis is performed by subdividing the frame into smaller chunks of data that we call macroblocks. Each of these macroblocks is then individually analyzed to determine whether it should utilize spatial redundancy, i.e., be intra coded, or utilize temporal redundancy, i.e., be inter coded. Intra coding relies only on redundant information within the current frame, while inter coding utilizes redundant information within previously coded frames.

This analysis is performed by the *motion estimation (ME)* and *choose intra prediction* components and as part of the process each mode and technique described in sections 2.2.3 and 2.2.3 are applied to each macroblock. The coding efficiency of each mode and technique are then compared, commonly by using a sum of absolute differences (SAD) or a sum of absolute transformed differences (SATD). As mentioned in the introduction to H.264, the standard only encompasses the specification of a valid output stream and the subsequent decoding of such a stream, it is therefore determined by the encoder implementation what encoding type and modes to use for each macroblock. An encoder implementation could therefore simply encode all macroblocks using a predefined intra mode, for example, but this would also mean that the coding efficiency of this encoder would be very poor. As such, finding the best possible encoding strategy for each macroblock is vital. This naturally requires testing a large number of modes for a large number of macroblocks, per frame, reducing the time it takes to encode a high-quality video stream is therefore of vital importance. As such, the benefits of parallelization are apparent already at an early stage of our review of H.264.

Moving on, the result of the analysis stage is a predicted macroblock P, encoded using a

mode that was deemed the best fit during the analysis stage. In *intra* mode, *P* is created solely from reconstructed samples in the current frame F_n. Using reconstructed macroblocks instead of the original input ensures that the encoder and decoder do not drift, which can introduce errors that will propagate over time and increase in severity and lead to a reduced video quality. In *inter* mode, *P* is formed by a *motion-compensated (MC)* prediction, from one or more reference frames, seen as F'_{n-1} in figure 2.7. Depending on the type of inter prediction, *P* is predicted from either one or two previously encoded, decoded and reconstructed frames.

The prediction *P* is then subtracted from the current macroblock to produce a residual or difference macroblock D_n. This residual block is then transformed in step *T*, using a block transform. In H.264 this block transfer is an integer spatial transform, which is an approximation to Discreet Cosine Transform (DCT). This transformation is applied to 4x4 blocks of pixels and is used to compact the coefficients into the low frequency domain, i.e., cut off high frequency outliers. The transform coefficients from *T* are then quantized *Q*. Quantization is the only stage during the encoding where data compression takes place, i.e., where data loss occurs, and subsequently the quality of the stream is reduced. This occurs, because high frequency data is removed from the residuals thus reducing the entropy of the residuals. This is also the lossy step in most image compression formats. The quantization step results in a set of quantized transform coefficients *X*.

These quantized transform coefficients then are used in two separate paths. Where the one prepares them for transmission or storage. In this case, they are re-ordered and entropy coded using either Context Adaptive Variable Length Coding (CAVLC) or Context Adaptive Binary Arithmetic Coding (CABAC). CABAC results in a better compression ratio, but at the cost of higher computational complexity. Once encoded, the coefficients, together with additional information required to decode the macroblock, such as the prediction mode (I,B,P), quantizer step size, motion vector information (if inter predicted) is passed to the Network Abstraction Layer (NAL) for transmission or storage.

The second path uses the quantized macroblock coefficients *X* in order to reconstruct a frame for encoding of further macroblocks. The coefficients *X* are re-scaled in Q^{-1} and inverse transformed in T^{-1} to produce a difference macroblock D_n'. This is not identical to the original difference macroblock D_n.

The prediction macroblock *P* is added to D_n' to create a reconstructed macroblock uF'_n, which is a distorted version of the original macroblock. In the case of intra coding, uF'_n is passed directly to the intra coding stage. For inter coding, uF'_n is passed through a de-blocking filter, which is applied to reduce the effects of blocking distortion. Finally, a reconstructed frame F'_n is created from a series of macroblocks and can then be used as a reference frame F'_{n-1} for inter coding.

H.264/AVC Decoder

By comparing the encoder in figure 2.7 to the decoder in figure 2.8, we can see that the decoder is basically contained within the encoder. The purpose of reconstructing the frame in the encoder is to ensure that both the encoder and decoder use identical reference frames to create the prediction *P*. If this is not the case, then the predictions *P* in encoder and decoder will not be identical, leading to an increasing error or drift between the encoder and decoder.

So, to briefly summarize, the decoder receives a compressed bitstream from the NAL. The data elements are entropy decoded and reordered to produce a set of quantized coefficients *X*.

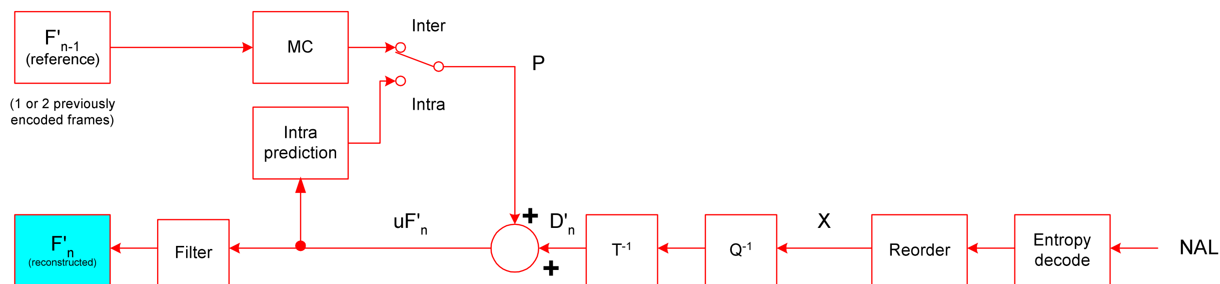


Figure 2.8: H.264/AVC Decoder, figure by I. Richardson [29]

These are rescaled Q^{-1} and inverse transformed T^{-1} to give a difference macroblock D_n (this macroblock is identical to D_n in the encoder).

Using the header information (motion vectors, etc), a prediction macroblock *P*, identical to the original prediction *P* formed in the encoder, is created. *P* is added to D_n' to create a reconstructed macroblock uF'_n, which is a distorted version of the original macroblock. This one is passed through the de-blocking filter, to reduce the effects of blocking distortion and a reconstructed frame F'_n is created from a series of macroblocks, resulting in a decompressed video sequence.

Data decomposition

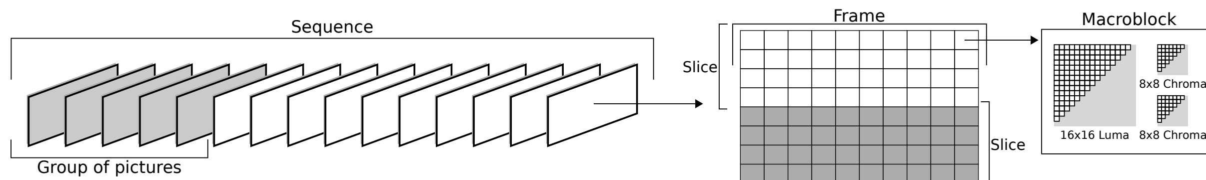


Figure 2.9: Overview of a video sequence

Looking at figure2.9, we can see how a H.264 video sequence can be decomposed. As we can see, a video sequence consists of a continuous stream of frames, which are divided into groups of pictures (GOP). A GOP is an example of a partition within the video stream, as such, the video encoding process is delimited by the boundaries of these GOPs, and they can therefore be distributed across multiple servers in a distributed system, if required. Each GOP, in turn, contains a number of frames, where a frame is further divided into one or more slices. In H.264, a slice functions as a spatial boundary, so if a frame is divided into multiple slices these are considered independent components. This provides a parallelization opportunity, as each slice within a frame can be encoded and decoded in parallel (as long as any potential reference frames already exist in the frame store).

A slice is in turn divided into fixed size macroblocks that each cover an area of 16x16 pixels of the luma component and a smaller 8x8 sample of the two chroma components. This sampling scheme is commonly referred to as the YC_bC_r color scheme or *4:2:0* chroma format [30]. Subsampling in this manner is common, because the human visual system is more sensitive to changes in luminance in a scene than to changes of chrominance. Depending on the encoder implementation, either the luma or chroma components can be used when determining what coding format and mode to use for a macroblock.

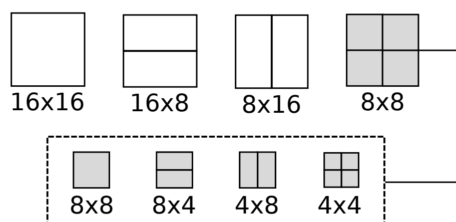


Figure 2.10: Macroblock partitions

Though the fixed size macroblock partitioning of 16x16 is utilized, a salient feature of H.264 is the introduction of variable block sizes, as seen in figure2.10. During motion estimation and intra prediction it is then possible to further subdivide a macroblock for the purpose of more accurately predicting spatial and temporal redundancy.

As we can see, H.264 is a workload that consists of sub-dividing the input stream and applying an evaluation to those sub-components, thus providing a number of data decomposition opportunities. Due to the layered approach, we also get nested decomposition opportunities:

Listing 2.1: Nested decomposition

```
1 for( frame in stream ):
2     for( slice in frame ):
3         for( macroblock in slice ):
4             for( subblock in macroblock ):
```

Returning to the original discussion, utilizing a successive approach by applying one or more operations to each distinct data unit in turn implies that all these operations occur sequentially. Then, if there are multiple processing units available, one is not able to utilize the parallel resources that are available, thus increasing the overall running time of the application. Expressing these data decomposition opportunities in a way that makes it possible to parallelize data operations can therefore increase performance considerably.

Functional decomposition

As we saw from figure2.9, a frame can consist of multiple slices. Each individual slice can be an *Intra predicted (**I**) slice*, an *Inter predicted (**P**) slice* or a *Bi-predicted (**B**) slice*. In addition, there exists *SP* and *SI* slices, which are *switching* P and I slices that make it possible to change between quality layers within a stream during playback, though we will not discuss those here.

I slices can only be intra coded, i.e., take advantage of spatial redundancy within the slice. The first frame in a GOP typically consists of only intra coded slices or macroblocks. This makes sense, since the implication is that the frame does not depend on references to other frames.

P slices can contain I macroblocks and predicted (P) inter coded macroblocks. With P macroblocks, temporal redundancy is exploited by searching through a maximum of 16 previously coded reference frames during motion estimation for the purpose of locating the movement of the macroblock between the frames. The motion estimation search results in at most one motion compensated signal per prediction, i.e., one motion vector. Motion estimation can result in a very large search area.

Finally, we have B slices, where the macroblocks can, in addition to being I or P predicted, be bi-predicted (B). The salient difference between a P and B predication is that the B macroblock results in two motion compensated signals (two motion vectors), where the weighted average of their content is used during motion compensation.

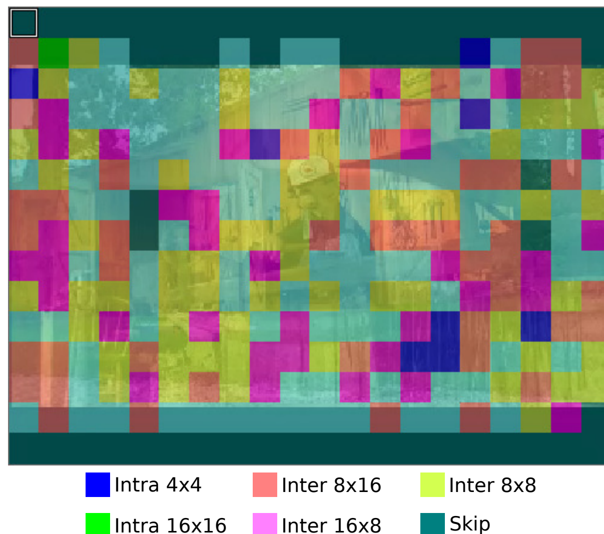


Figure 2.11: Macroblock sub-partitioning into blocks

To exemplify, in figure2.11, we have a frame with one P slice and the subsequent coding types of the macroblocks, which in this case are restricted to I and P macroblocks. As we can see though, there is one additional type, a *skipped* macroblock. A skipped macroblock contains a zero-length motion vector, with the implication being that the macroblock has not been displaced, and therefore contains the same content as previously. We will discuss this further in section 2.2.3 that focuses on inter prediction. First, however, we will describe how intra prediction is performed.

Intra prediction

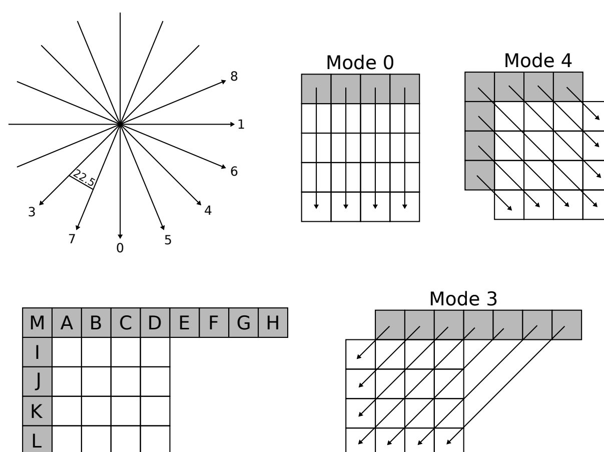


Figure 2.12: Intra prediction modes

As previously described, an intra coded macroblock makes use of spatial redundancy within

a slice and therefore holds no references to other frames. Intra coding is supported at a variety of granularities, from 16x16 to 4x4 luma and chroma blocks. A larger size block is better suited for predicting smooth areas with less detail, while smaller blocks are more useful for predicting irregular, highly detailed areas. The reason for this, is because spatial prediction is based on copying the luma or chroma data from its neighboring pixels, doing so using one of the 8 (mode 0-1 and 3-8) directional predication modes seen, as illustrated in figure2.12. These directional modes simply copy the corresponding pixel values of A-M into their respective locations in the predicted block. The luminance of a block of pixels is then predicted by copying the color-space directly above, to the left, right, etc. In addition, there is mode 2, the DC mode. DC-prediction, calculates the mean of the four pixels above and the four pixels to the left of the 4x4 pixel block and uses this to paint the block of pixels. Depending on the size of the block all of these modes are not available, for a 16x16 luma block, only modes 0, 1, 2 or the plane mode are available. In addition, there is an *I_PCM* mode, which skips the prediction and transform stages. This mode is used for macroblocks where no previously coded macroblocks are available.

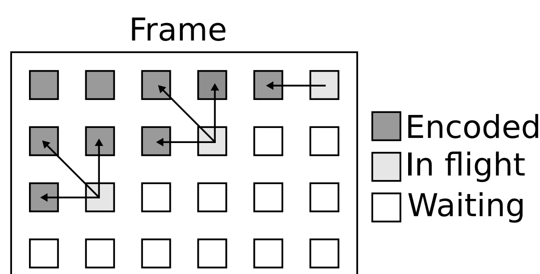


Figure 2.13: Macroblocks, wavefront dependencies

During intra prediction, these modes are tested for their accuracy. Depending on the size of the slice and number of blocks, the number of predications to attempt can become quite large. Since there are no dependencies between the application of these predictions, except that that the data in the pixels A-M needs to be available, as illustrated in figure 2.13, these also provide a parallelization opportunity. This also illustrates how multimedia algorithms typically involve both data and task parallel opportunities. In this case, we have a task parallel opportunity, which arises from the possibility of applying multiple modes in parallel. Simultaneously, we have multiple macroblocks that are ready for coding simultaneously. Furthermore, we are starting to see the contours of the complex data dependencies that can arise from complicated workloads such as H.264.

Inter prediction

With inter prediction temporal redundancy between successive frames is exploited, as seen in figure 2.14, since there is a high probability that there will be considerable similarity between them, i.e., objects moving in a scene will be displaced by some factor, but will still be present in subsequent frames. Inter prediction is then achieved by predicting the motion of objects in the current frame from previously coded frames, also known as reference frames. This process of determining the displacement of a macroblock, in relation to its reference frames, is called motion estimation. The result of this motion estimation is a motion vector, which is represented as a displacement of pixels, for example (2, -3), i.e, the macro-block can be located in the reference frame using this displacement factor.

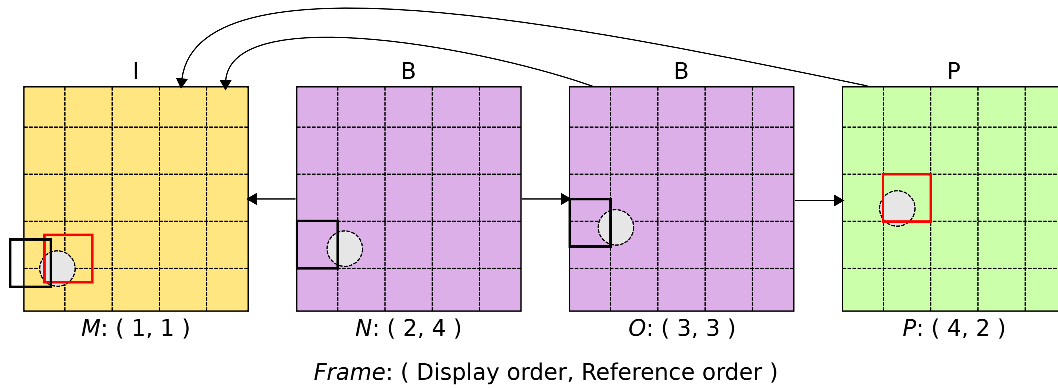


Figure 2.14: Inter prediction

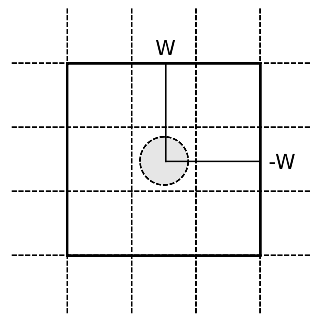


Figure 2.15: Motion estimation, search area

When determining the motion of an object, the area to search for it in subsequent frames is limited to a certain length in pixels, as seen in figure 2.15. With a frame rate of 30 FPS, the probable displacement of an object is quite limited, as it is unlikely that an object has moved very far between successive frames. There exists a number of search algorithms, such as full, three step, diamond and more. However, given the size of this search area, and with a full search algorithm, i.e., all possible starting points are used to search for the object, the number of positions to search in can become extremely large. This motion estimation search can, however, be parallelized, but represents one of the most time consuming aspects of the encoding process. Particularly when we consider that H.264 can search in a maximum of 16 previously coded frames.

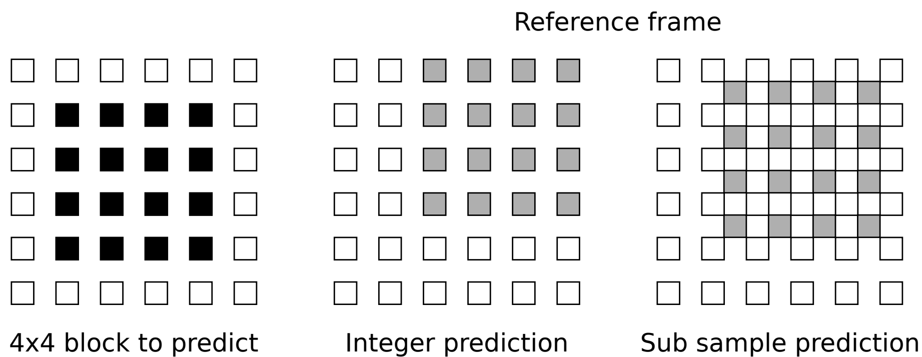


Figure 2.16: Sub sample prediction using interpolation

Furthermore, H.264 allows for sub sample motion compensation, through the use of interpolation, as seen in figure 2.16. This subpel accuracy can improve the accuracy of the estimation, as integer accuracy implies some loss. As such, interpolation can increase the resolution of the image, which is achieved by, for example, doubling the pixel space, and adding pixels between two adjacent pixels and taking the uniform color space between them. In H.264, interpolation can improve accuracy up to a quarter space resolution. While this can improve the quality of a displayed frame, because the prediction is more accurate, it also increases the complexity of the pipe-line and the search area.

Another complicating feature in H.264 is the possibility of using variable block sizes. As we can see in figure2.10, H.264 allows for varying size in the size of the predicted blocks, down to a block size of 4x4 pixels. This makes it possible to predict the motion of areas with high-levels of detail more accurately. In figure2.11, we can see how the motion estimation process can result in a variation of sizes of predicted blocks. However, this also adds to the computational complexity of the analysis stage.

As we can see in figure 2.14, we have four frames in display order. Each containing one single slice, coded as *I, B, B, P* respectively. Frame *M* is intra-coded, since it is the first frame in our GOP, and also is the first in the reference order. Frame *P*, which is an inter predicted P frame, is last in the display order, but second in the reference order. This highlights another feature of H.264, that the reference order is decoupled from display order. With the implication that frames can be encoded or decoded in a different order than they are displayed. This ability is a feature of H.264, which makes sense from a coding perspective, but can serve to confound an application developer who wants to parallelize these aspects. Some method of determining what frames depend on a subset of data then has to be in place, before executing the task for decoding a given frame.

Going back to figure 2.14, we can see how the red highlighted macroblock in frame *P* uses frame *M* as a reference frame. This red macroblock is a predicted (P) macroblock and therefore only contains one motion vector describing its displacement relative to its *one* reference frame, *M* in this case.

The black macroblock in frame *N* is a bi-predicted (B) macroblock, as such it can contain two motion vectors and therefore two reference frames, these references can both point to the same frame. In this case, the black macroblock in *N* is predicted from references to frames *M* and *O*. As such, this macroblock can only be decoded once all its reference frames have been decoded. As such, even though frame *N* is second in the display order it can only be decoded as the fourth frame, due to the references it holds.

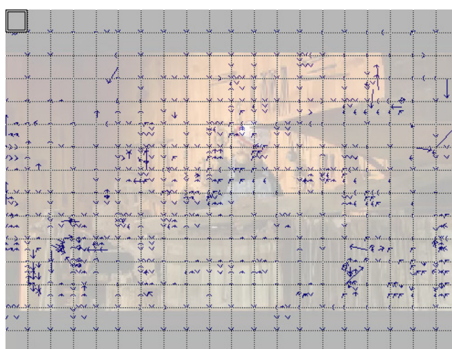


Figure 2.17: Macroblock motion vectors

In figure2.17, we can see how the resulting motion vectors after a motion estimation pass has been performed look like. As we can see, parts of the image with high detail and more motion have a higher compaction of motion vectors. We can also see that most of them are relatively short in distance.

While there exits a number of dependencies between macroblocks in *intra* prediction, *inter* prediction complicates things further, as a macroblock can now be predicted based on one or more reference frames. What is evident from figure 2.14 is that the dependencies between frames can become very complicated. Particularly when considering that a single macroblock of 16x16 pixels, potentially can contain a maximum of 16 motion vectors. Furthermore, all these potential inter-dependencies can cause problems when working on shared data structures, particularly if the same memory addresses are supposed to be updated as it could possible cause contention.

In addition, it is also becoming clear that there is an extreme number of different modes to test at each stage, to determine the best encoding type for a given macroblock. Furthermore, the implementation complexity is extreme, as we will get a better feeling for in the following sections.

Application complexity

Managing application complexity can be a daunting task, particularly when faced with an application such as H.264. In figure 2.18, we can see a call graph of the *x264* [31] encoder, which is an open source implementation of the H.264 video standard.

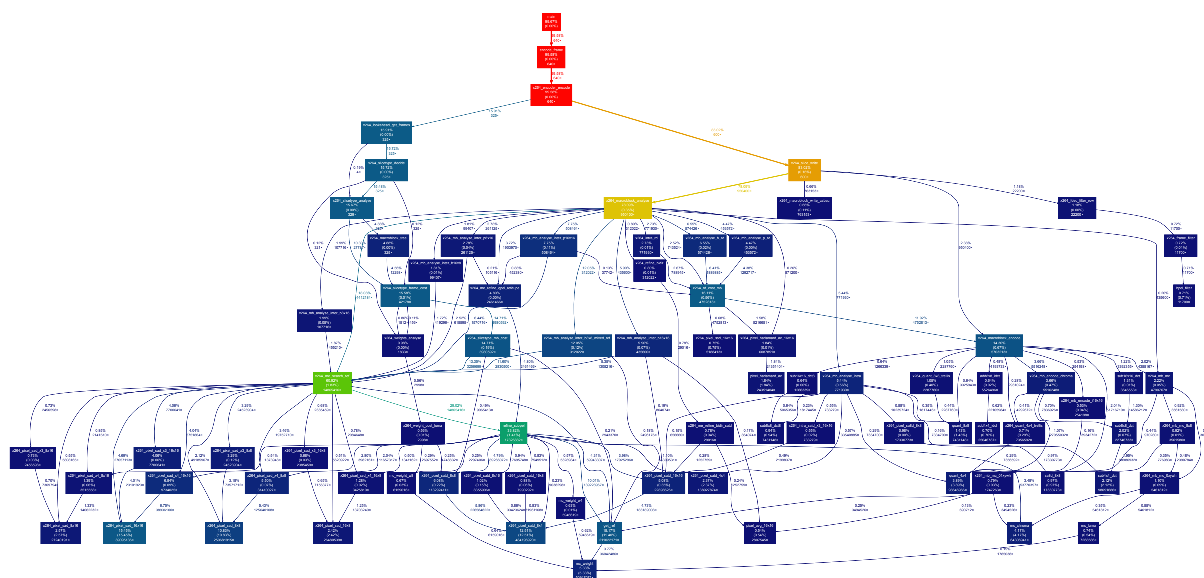


Figure 2.18: x264 call graph

Initially in this chapter we discussed how a decomposition tree can aid in understanding the requirements of an application, this becomes more apparent when we consider the x264 call graph at hand. What is also apparent is that creating a parallel workload of H.264 is a non-trivial task, and therefore developing tools that can aid this process is vital.

2.3 Conclusion

Interactive multimedia workloads, to a varying degree, depend on an element of real-time interaction. As such, they are bound by strict requirements to timeliness. If an event, a processed image or a encoded frame do not arrive within their deadline, they are effectively of little use. Given the computational requirements of these workloads, relying on sequential resources is no longer a viable approach. At some level, multiple hardware resources need to be coordinated in order to achieve an acceptable Quality of Experience (QoE) for the end-user. Real-time processing of a video stream and subsequent encoding to H.264 would require an extremely fast uniprocessor, hardware encoding or parallelization. Consider encoding a video stream at 30 frames per second (FPS) at 1920 x 1080 resolution. For each frame in the sequence, we then have 33 milliseconds to encode it. That is a minimum of 8040 macroblocks to evaluate. Decomposing or partitioning the application then provides a viable approach to solving these issues. However, to accomplish this, we need to understand how the application works. As we have seen with H.264, these workloads can be immensely complex, and encompass a wide variety of functionality. As such, a major task is to evaluate what we have learned by examining these applications and how this can be used as input to methods for more readily supporting the implementation of interactive multimedia applications utilizing distributed hardware resources.

3 | Parallel Programming Model

A parallel programming model is the abstract definition of conceptual ideas for structuring parallel programs. The existence of parallel programming models comes as a result, among other, of the inherent difficulty of dealing with nondeterminism and synchronization of shared state. While using a high-level parallel programming model is easier for developers to work with and understand [32], writing a parallel application, when compared to its serial counterpart, is nonetheless still harder [33].

While existing models vary greatly in their functionality, frequently limitations exist in their application to certain application domains as no general solution exists. Furthermore, a number of the existing models sacrifice functionality or forms of parallelism that are not required for the domain of applications it is designed for. In this chapter we will consider the characteristics of interactive multimedia applications, and their subsequent workloads as described in chapter 2, and consider how the relationship between parallelization and application domain impact the high-level parallel programming model. Where the focus of the analysis is on how to fashion a cognitive model that can aid a developer with efficiently implementing interactive multimedia applications in parallel execution environments. To achieve this we will consider this relationship between application domain and programming model in terms of qualities and concepts, and discuss how to best provide a delivery system for such functionality to the developer.

In the previous chapter we discussed parallelization opportunities within interactive multimedia application workloads. In this chapter the focus is moving towards how to exploit the parallel nature of such a workload. A natural starting point is then Ian Foster's model for understanding how to design parallel algorithms.

3.1 Designing parallel algorithms

In [34], Ian Foster has succinctly described a methodology for designing parallel algorithms, using the process illustrated in figure 3.1. As our primary concern is to create a parallel programming model, understanding this process can provide valuable input to its design.

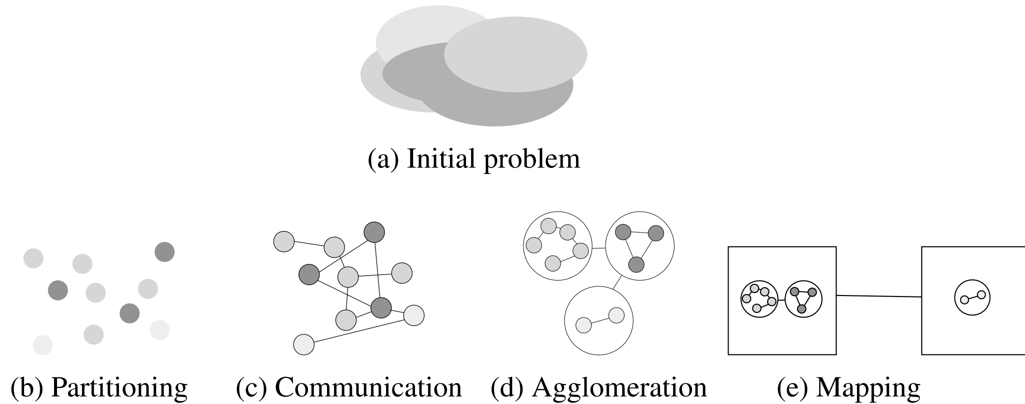


Figure 3.1: Ian Foster's stages of designing a parallel algorithm

Following figure 3.1, the process unfolds as follows, given an initial problem, as seen in figure 3.1a, the first stage is to partition the problem into its constituent parts, which is accomplished by decomposing the computations that are going to be performed and the data they work on into independent tasks, as seen in figure 3.1b. At this stage, it is important that we create as fine a granularity of tasks as possible thereby exposing as much of the parallel nature of the application as possible.

After we have partitioned the problem into smaller components, we then need to determine what communication is required between the partitioned components to coordinate the execution of the workload, seen in figure 3.1c. As such, communication consists of discerning the flow of data and dependencies between the tasks.

The *partitioning* and *communication* stages consist primarily of specifying the interaction patterns of the application itself. During these stages we do not attempt to optimize for the execution of the application, we are more concerned with being able to express as much of the parallel nature of the application as possible in as fine a granularity as possible.

Moving on, we come to the agglomeration stage, as seen in figure 3.1d. During agglomeration, we evaluate the results of the previous two stages by looking at the fined grained components and their communication patterns with the purpose of determining if combining tasks can reduce communication or computational overhead. One could also agglomerate tasks to reduce implementation complexity, even if doing so would reduce the potential speed-up of the application.

Finally, during the final stage of mapping, as seen in figure 3.1e, we take the refined result from the agglomeration stage and, based on the available hardware resources, map the individual tasks in the application to those resources.

The *agglomeration* and *mapping* stages are thereby more concerned with optimizing the application given a hardware topology. This makes sense as different hardware architectures can have greatly varying properties. If communication between two processors is expensive, co-locating tasks that communicate frequently makes sense, or perhaps even combining the tasks into one component during agglomeration.

Given these stages of implementing a parallel algorithm, the purpose of a high-level parallel programming model, and its subsequent implementation, is then to automate or alleviate one or more stages of designing parallel algorithms. This can be achieved a number of ways, such as by ensuring that one or more of the dependent tasks (communication, scheduling, etc.) for achieving parallel execution of a workload are abstracted away from the developer. Primar-

ily, the developer should be supported, as much as possible, in the process of describing the workload in terms of task partitioning and describing communication patterns or data flow.

In the remainder of this chapter, we will therefore review a number of desirable qualities such a model should exhibit, followed by a discussion of concepts from existing models that would be preferential to have support for, and finally, we will discuss how to enable developers to interact with it. In so doing, we will shed some light on the progress made so far in the support of parallelizing multimedia algorithms.

3.2 Qualities

There exists a number of qualities one would expect a parallel programming model to exhibit [35]. In this section, we will briefly describe some of the most important qualities a high level parallel programing model should exhibit. In this discussion we do exhaustively examine all qualities a programming model might consist of, but rather focus on aspects that are important to this thesis, such as the fact that these applications are intended to run in a parallel execution environment, and that from a developer's perspective ease of use is very important. For convenience, we have loosely grouped these different qualities into three groups according to their overall contribution to that area.

3.2.1 System

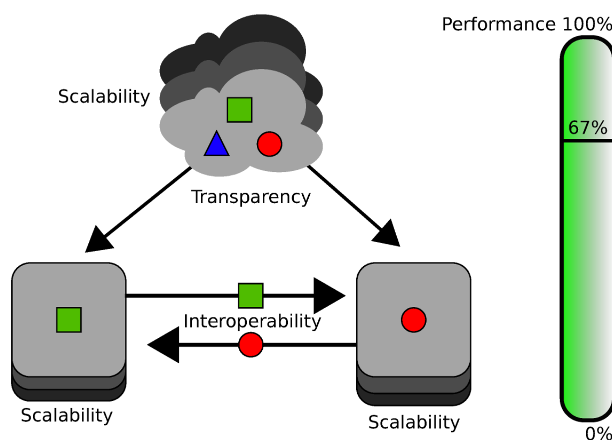


Figure 3.2: System qualities

The following qualities are closely related to the high-level programming model's relationship with the underlying hardware resources, or more specifically, with how we can efficiently conceal those resources from the developer. As such, these qualities are concerned with finding the balancing point between ensuring transparency, while maintaining performance, scalability and interoperability.

Interoperability This quality refers to the ability of multiple systems or components to exchange state and subsequently process that state for the purpose of progressing workload execution. Interoperability occurs at a number of levels within a workload. Two distinct hardware

architectures are said to be interoperable for a given workload if they are able to freely migrate tasks between each other. Two tasks are said to be interoperable if they are capable of coordinating and synchronizing state during execution when they are implemented in different programming languages. From a programming model perspective, interoperability can be assured through a combination of decoupling components, well defined interfaces and standardized formats. Supporting interoperability is important, because it can potentially ensure that future hardware architectures or programming languages are adapted for use within the context of the programming model.

Scalability A scalable system is one that is capable of handling continuous increases in work. Scalability can be accomplished at different levels, for loosely coupled systems simply adding additional hardware resources can ensure that we are capable of handling all incoming requests. In a number of circumstances this approach is not sufficient, but must be coupled with a scalable run-time system or in some cases even algorithm. A scalable run-time system is one that remains suitably efficient and practical, even when operating under heavy load, during coordination of a large number of tasks or when handling a large number of hardware resources. A scalable algorithm is one that operates well even when faced with a complex scenario or a large input set. Having a scalable system is therefore important in order to ensure the efficient execution of large numbers of parallel tasks, and their interaction and communication.

Transparency A transparent programming model is one that is capable of concealing low-level system details from the developer's view. This is accomplished by creating abstractions that make it possible to divert low-level task handling, such as synchronization, distribution, communication, load-balancing, fault-tolerance, scheduling, etc., to compile-time or to an underlying run-time system. The difficulty then lies in finding a trade-off between concealment of low-level tasks while simultaneously ensuring that a sufficient amount of information is supplied so that tasks can be efficiently executed. Transparency can also be beneficial with regard to the obsolescence of hardware architectures and emergence of new platforms. Appropriate programming model abstractions can conceal the details of the underlying architecture of the hardware, while still support the utilization of heterogeneous hardware, such as multi-core CPUs, GPUs and future hardware architectures or communication infrastructures.

Performance The purpose of a high-level parallel programming model is to ensure that the trade-off between ease of implementation and efficiency has been met. The programming model should therefore ensure that the developer is able to sufficiently define their application in such a way that tasks can efficiently be scheduled and executed and that data is correctly and timely disseminated, and that all this occurs within the performance requirements of the application. After all, if the application is not capable of running according to its performance requirements, there is little point in having a convenient way of structuring the application.

3.2.2 Maintenance

The following qualities are concerned with the ability of the application developer to maintain the software over time, and forms an important part in supporting the application throughout its life time.

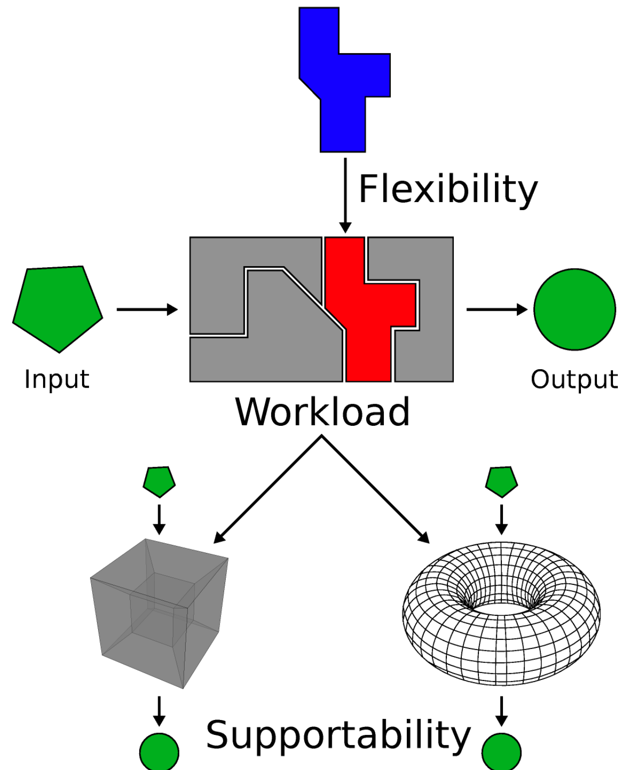


Figure 3.3: Maintenance qualities

Supportability This quality can encompass a range of functionality, such as monitoring system state, reconfigurability, etc., and while these properties also are of interest we are more concerned with the ability to debug the system. An implication of concurrent execution is non-deterministic behavior, i.e., the execution order of tasks between runs is not necessarily the same. Small differences in timing can lead to a reordering of execution. This inability to guarantee the order of execution between runs also means that reproducing errors, with the intent of fixing the system, is nearly impossible. To this extent, having some ability to reliably reproduce faults is perhaps one of the most desirable qualities a parallel application can have as it means that fixing bugs in the system is considerably easier.

Flexibility Flexibility is the degree to which the code written by a developer is applicable in different execution environments. Flexible code then implies that the developer can write the code once, and have it execute on a variety of existing and future hardware architectures. To achieve this flexibility, a general prerequisite is that a sufficient abstraction between application code and hardware exists, where the essential part is to ensure sufficient decoupling from the processing language to the execution environment. The benefit of flexibility is the applicability of the software for future architectures.

3.2.3 Development

A software development process consists of a number of stages that can be configured according to the requirements defined by the project at hand. The stages that are included are typically *requirements*, *specification*, *architecture*, *design*, *implementation*, *testing*, *debugging*,

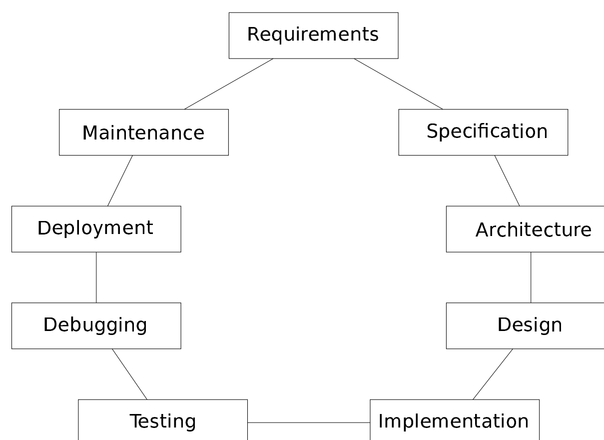


Figure 3.4: Software development process

deployment, *maintenance*, where a model such as water fall, agile or similar, determines which stages are included in the process, how frequently they are revisited and how long time is spent within each stage. Where the focus of this thesis is on improving support during the design, implementation, debugging and maintenance stages. The following qualities are related to the development process, where these qualities are more abstract than the ones covered thus far and are closely related to the architecture and design stages of the application development process. Some of these qualities can have a clear impact on how long those stages might be, decreasing the time spent there means more time can be spent during debugging, maintenance, etc. It is fairly easy to agree whether an application is able to scale or is easy to debug, but more subtle to agree on what makes a developer productive or what can ensure adoption of a programming model. However, while we might not agree exactly on what constitutes productivity, it is nonetheless a desirable quality to achieve.

Productivity Productivity is a measure of efficiency of production. The purpose of this quality is then to ensure that the developer achieves their goal by making it possible to produce code efficiently and focus on the task at hand. There are a number of things that might negate productivity, such as complex language interaction, unclear semantics or large APIs, all of which should be avoided as far as possible. Supporting productivity can be achieved by deferring mundane and repetitive tasks, such as scheduling, communication, etc., to compile time, run-time systems or similar, thus allowing the developer time to focus on the implementation of their application.

Expressiveness As we have seen in chapter 2, interactive multimedia application workloads can be quite complex. As evidenced by H.264 the workloads can contain cycles, a myriad of data inter and intra dependencies and large numbers of interacting tasks. To this regard, it is fundamentally necessary to be able to accurately model all types of relationships and constructs found within the applications. This is important, because an expressive programming model is capable of modeling the entirety of the application, covering all forms of parallelism, deadlines, cycles or other constructs that are required to correctly model the application.

Usability This refers to the ease of use and ability to learn the programming model. Within the context of the stated application domain of interactive multimedia applications, this implies that it should be easy for the developer to model the application using the language constructs given, and that learning how to correctly use those language constructs does not cause excessive overhead. A developer with some familiarity of the characteristics of multimedia workloads should be capable of easily deducing the intentions of the model with some ease. This is important, because an overly complex model is not usable.

Correctness The functional correctness of an algorithm is asserted when the algorithm is correct with respect to a specification, i.e., for each input the correct output is produced. As such, the programming model should ensure that, when given a functionally correct implementation of an algorithm, that the expected output is produced. This is an important feature, because it makes it possible to more easily make assumptions about the correctness of subcomponents of a system, which in turn can facilitate compositionality, i.e., the ability to compose a system of multiple components.

Adoption Adoption refers to the extent to which a new programming model is utilized. One of the primary obstacles for adopting a new programming model is lack of familiarity. As such, it is essential that concepts and language constructs familiar to the developer are utilized, i.e., it should be possible to use existing high-level languages when writing code, thus enabling the use of familiar development tools, debuggers, etc. Ensuring adoption is important, because it can lead to a vibrant developer community, which in turn can lead to the development of third-party utilities, such as Integrated Development Environments (IDEs), debuggers, etc.

3.2.4 Summary

We have so far covered a number of different qualities that are important for a high-level parallel programming model. Some more or less concrete and others that complement each other or stand in opposition. Usability and adoption might be complementary, while expressiveness and usability might be more opposing. Finding the correct balance between these qualities is then a major concern, and can be difficult process to complete, which will require some compromise. As we have seen frequently, a number of programming models optimize for performance in a given application domain by catering to certain application characteristics, or limit the model to certain tasks, but in doing so; sacrifice some generality in the applicability of the model in other application domains. Finding these trade-offs and making the right choices is highly demanding. Given the qualities we have described, we will, in the following sections, consider what concepts and ideas can ensure that a high-level parallel programming model achieves such as balance, given the application domain of interactive multimedia applications.

3.3 Concepts

There exists a number of abstract concepts that can provide useful input to the formulation of a high-level parallel programming model. In this section we will examine some of the central concepts found in the imperative and declarative programming paradigms and how they relate

to the interactive multimedia application domain, as described in chapter 2. Looking at the concepts that compose existing paradigms, and the relationships formed between these concepts, can help us formulate new languages with new capabilities, as described by Van Roy et al [36]. As such, the purpose of this evaluation is to determine what combination of concepts can support interactive multimedia application development, given the desirable qualities described in section 3.2, when operating in a parallel execution environment.

The imperative and declarative programming paradigms utilize different concepts to formulate their operational semantics and programming languages. These differences in approach have an impact on the application developer's interaction with the code, but also has an impact on how the code is executed during run-time. Commonly, these two forms, imperative and declarative, are considered opposing models, where a declarative language is designed in such a way that the application developer can focus on describing *how* they want to accomplish a task, i.e., how something is to be evaluated without specifying control flow, while declarative languages are concerned with facilitating the developers in stating *what* they want to accomplish, i.e., providing a step by step description of what is to be computed and when. As such, the language constructs in the imperative paradigm were, and still are to some extent, more closely linked to the structure of the underlying hardware.

Imperative programming languages, such as C, C++, Java and C# have dominated the application development landscape [37] for decades. It is not the purpose of this thesis to discuss the reasons for this dominance, as they may be many, but contributing factors have been the fact that imperative programming languages typically have resembled the underlying hardware architectures, which has made it easier to design efficient applications, and furthermore, the speed-up of sequential processors has followed Moore's law [38], which has guaranteed a continuous increase in sequential performance, which has allowed applications to scale as required. However, with the introduction of multi-core CPUs, the hardware landscape has changed, and parallel execution is now required to achieve the same scalability [39]. With concurrency, however, nondeterminism appears naturally, and due to the frequent use of shared named state and code with side-effects, multi-threaded execution is something imperative languages inherently handles poorly. Primarily, because the programming paradigm no longer maps readily to the available hardware resources.

The reason why imperative languages do not deal well with concurrency, is due to their extensive use of shared named state. This causes problems because two threads executing within a process share the same address space, thus, if they access a common variable simultaneously, small differences in timing can cause the operations on that shared variable to be reordered, resulting in vastly different results between runs of execution. This is called observable nondeterminism, i.e., that a user can see different results from an execution starting with the same initial configuration, where the result of the program depends on these differences in timing between the execution of different parts of a program, also called a race-condition. As such, writing parallel applications in imperative languages is inherently difficult because a developer has to correctly limit the impact of side-effects caused by state-full objects and shared named state, using low-level synchronization mechanisms. As such, the *productivity* of the developer has greatly decreased as a result of the emergence of concurrent execution in the imperative programming model domain. Nonetheless, the imperative paradigm is by far the most widely used, and to ensure adoption, supporting this paradigm is highly useful. Though we will also see that there are other favorable properties to be found in the imperative programming paradigm.

Declarative languages, and functional programming languages in particular, make extensive

use of referential transparency, side-effect free execution through pure functions, and do not have named state. Furthermore, a number of decisions with regard to control flow and implementation specific details are deferred to compile time, unlike with imperative languages, where the application developer typically operates at an abstraction level closer to hardware and thus has finer grained control. Referential transparency means that an expression can be replaced with its value without changing the behavior of the program. Side-effect free execution means that when an expression is evaluated it does not modify any state or have observable interaction with other components in the system. No named state means that there is no shared state that can be accessed and modified simultaneously. In combination, these properties means that it is a lot easier to execute a declarative program in parallel execution environment, because the most difficult aspect, of keeping consistent state during concurrent execution can be handled during compile time. As such, the application developer does not need to be concerned with such low level details. This implies that the application developer can be more productive in a parallel execution environment when writing in a declarative style. However, as a number of the low-level details are concealed there can also be some loss of performance. There are also issues related to interactive multimedia workloads, as we saw in chapter 2, the data dependencies are numerous and complex, and describing such interactions without named state is difficult, if not even impossible.

From the perspective of parallelizing interactive multimedia applications, neither imperative or declarative languages are completely compatible.Imperative languages are inherently designed for describing algorithms and have named state, which both match the multimedia domain. Parallel application development, on the other side, is better supported in the declarative languages. According to Loidl et al [40], following the idea of declarative programming, the main task of a parallel programmer should be to specify what has to be evaluated in parallel and not how the parallel evaluation has to be organized. From the application developer perspective, focus should be on the decomposition of their problems into parallel tasks. Parallel programming is inherently harder than sequential programming.

Considering what programming paradigms to support with the framework, or whether a completely new framework should be developed is also worth considering. An important facet of such a programming model is to ensure its usability. As such, developing a completely new programming language might be excessive, as all language constructs such as addition, subtraction, etc., need to be implemented. Also is additional overhead for the developer who has to learn a new programming language.

3.3.1 Named state

In the context of a running application, state is a term used to describe all the information stored at a given point in time [36]. Named state then refers to shared and accessible state, while unnamed state implies that there is no way of directly accessing the current state of the application. In the imperative paradigm, the computation is expressed by statements that change the program state. In contrast, in the declarative paradigm the program describes the desired results, and does not specify changes to the state directly.

To get some idea of the differences between named and unnamed state consider the following two examples, where we are multiplying the numbers from 1 to 10 by 2. In 3.1, we can see that we are primarily focused on describing the interaction between the functions, while in 3.2 a majority of the code is concerned with defining data access patterns.

Listing 3.1: Unnamed state in Clojure

```
1 (map #(* % 2) (range 1 11))
```

Listing 3.2: Named state in C++

```
1 int values[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2 for(int i = 0; i < 10; ++i)
3 {
4     values[i] = values[i]*2;
5 }
```

Named state provides an advantage when working with multimedia algorithms, because it is a lot simpler to define the data access patterns. A disadvantage is that the state is then shared and in a parallel execution environment this means that the state needs to be kept consistent, even when operating in a multi-threaded environment.

3.3.2 Determinism

Due to the design of modern hardware architectures, concurrent execution of code has become the natural way of increasing application performance. To achieve efficient concurrency, it is common to run multiple threads of execution within a process, but when multiple threads of execution are interleaved, any access to shared state can lead to subtle differences in timing or scheduling may cause a reordering of accesses or modifications to the shared state, which in turn can lead to observable differences in the result, even when given the same initial configuration. We have illustrated this issue in figure 3.5, where we have an original piece of sequential code and two different inter leavings of the code during concurrent execution. As we can see, due to a reordering of the interaction with the shared variable x, its final value is *1* in interleaving #1 and *2* in interleaving #2.

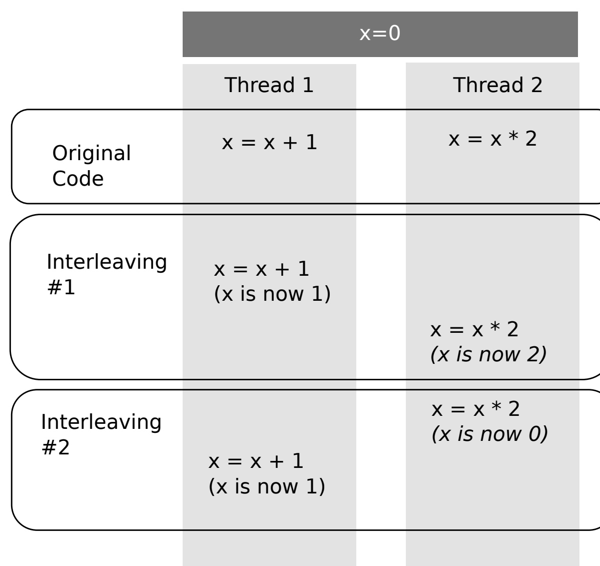


Figure 3.5: Interleaving of tasks during sequential and parallel execution

Regions of code where shared state is accessed is commonly called critical regions. As we can see in figure 3.5, we do not attempt to negate the effects of accessing shared variables during concurrent execution, but it is possible to minimize the effects of accessing shared state by using low-level locking mechanisms, such as mutex', semaphores and barriers. These locking mechanisms can be used to control the access to these critical regions, by serializing access to the shared state, waiting for a pre-condition to be met before allowing access, and etc. While these locking mechanisms make it possible to negate the effects of concurrent access to shared state, there are still some issues associated with them, as they are widely perceived to be very hard to use correctly. If used incorrectly, they can cause dangerous side-effects, such as starvation, where a thread of execution never gets to progress due to its low priority, or deadlocks, where the program will intermittently stall completely.

What we have described so far is commonly referred to as observable nondeterminism, and nondeterminism is a major sources of error in parallel computing, as subtle programming errors can lead to unintended nondeterministic behavior and hard to catch bugs. In [41], Bocchino et al argue that all parallel programming models must ensure deterministic execution as it will remove this uncertainty making it a lot easier for the developer to debug his application and reason about the performance of their application. As such, some of the benefits of deterministic semantics is an improved ability to reason about the application, and the ability to easily debug and test the application, without having to use replays [42] or other advanced techniques.

Deterministic behavior can be achieved in a number of ways. In [43], Bocchino et al extend the Java programming language with a type and effect system for achieving deterministic parallelization of Java applications. The extensions to Java make it possible for the developer to give names to distinct parts of the heap, called regions, and specify the kind of accesses different parts of the program, such as read or write effects, have. The compiler can then check, using modular analysis, that all pairs of memory accesses either commute with each other, e.g., they are both reads, or they access disjoint parts of the heap, or are properly synchronized to ensure determinism.

Kahn Process Networks (KPNs) also define a formalism for interaction semantics that ensure deterministic execution. A KPN [44] has a simple representation in the form of a *directed graph* with *processes* as nodes and *channels* as edges. A process encapsulates data and a single, sequential control flow, independent of any other process. Processes are not allowed to share data and may communicate only by sending messages over channels. Channels are *infinite* FIFO queues that store discrete *messages*. Channels have *exactly one* sender and *one* receiver process on each end (1:1 communication), and every process can have multiple *input* and *output* channels. Sending a message to the channel always succeeds, but trying to receive a message from an empty channel *blocks* the process until a message becomes available. These properties define the *operational semantics* of KPNs and make the Kahn model *deterministic*, i.e., the history of messages produced on the channels does not depend on how the process execution order. Less restrictive models, e.g., those that allow non-blocking reads or polling, would result in non-deterministic behavior.

While deterministic execution is not required or even wanted by a number of applications, those that depend on correct and reliable execution within a system to ensure operational correctness can benefit greatly from it. Deterministic execution is also a desirable quality for those that want to reason about their system more easily in a parallel execution environment, as it makes debugging and other maintenance tasks a lot easier to perform.

3.3.3 Forms of parallelism

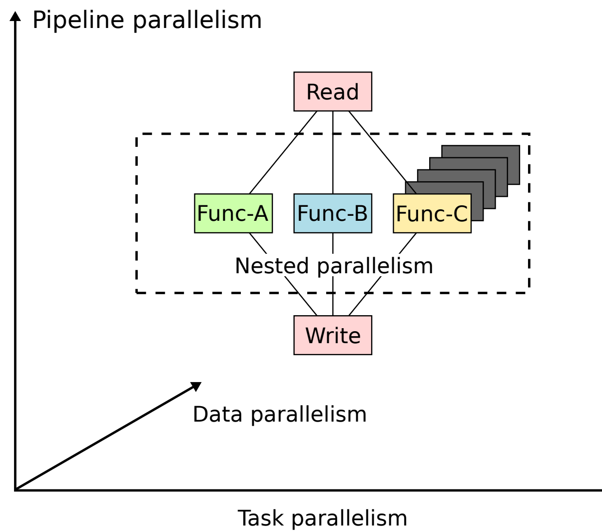


Figure 3.6: Types of parallelism

The first step in Foster's approach to designing parallel algorithms [34] is partitioning. This process involves decomposing the program into tasks and discerning how those tasks operate on the data they depend on for completing their purpose. This decomposition can then be achieved in two domains by expressing either task or data parallelism, where these are only two forms of known parallelism. There exists no formal terminology for describing the forms of parallelism an application can exhibit and there is no distinct set of types either. For the purpose of delimitation, we will focus on application level parallelism, which is the most relevant for the discussions in this thesis. We will therefore exclude bit or instruction level parallelism from these discussions. In essence, the types of parallelism we will consider are summarized in figure 3.6 and consist of *data*, *task*, *pipeline* and *nested* parallelism.

In this context *data* parallelism refers to a distinct piece of application code, such as a procedure, function or similar, applied concurrently to indistinct units of data. This can be viewed as a more traditional loop or for-each construct, i.e., for each element in this array apply this piece of code on it. In figure 3.6, data parallelism is shown on the z-axis, where we can see how *Func-C*, is replicated N times, which is to signify that given the input data from *read*, we can have N concurrent instances of *Func-C* operating on the data in parallel.

Task parallelism refers to indistinct pieces of code running in parallel, these tasks can operate on the same or distinct units of data. This can be viewed as a fork and join type of scenario, where the application execution is forked into multiple sub-processes, that perform distinct operations and whose execution is joined at a later stage of execution. In figure 3.6, this is shown on the x-axis, where we can see that *read* is invoking *Func-A*, *Func-B* and *Func-C* in parallel.

Pipeline parallelism arises due to continuous streams of data, loops, or continuous refinement, where the body of the computation consists of a number of subsequent tasks where each subtask depends on the output of the previous task to be run, effectively forming a pipeline of operations. In such a scenario, it is possible to parallelize the pipeline by filling up the pipeline with distinct units of data that can be processed at each stage in the computation in parallel. In

figure 3.6, we can see this illustrated on the y-axis, where *read*, *Func-** and *write* each can run in parallel given a continuous stream of input.

Finally we have *nested* parallelism, which occurs when one form of parallelism exists within another, task parallelism embedded with data parallelism, for example. In figure 3.6, we can see how all the *Func-**'s run in parallel simultaneously as *Func-C* runs in parallel over the distinct data units it was given as input.

A number of existing frameworks limit the ability to express either form of parallelism, which can make sense for certain classes of applications. Google's MapReduce [13] for example is a data-parallel framework and has no support for task parallelism, which is reasonable, as it was designed to operate on large sets of data. However, this also limits the applicability of the framework to support data parallel applications. Due to the wide popularity of this programming model task parallel extensions have appeared, such as Twister [45] and HaLoop [46], though these extensions have considerable overhead associated with them. Erlang [47] is a similar example, just for task parallelism, where Erlang is a functional programming language, which inherently models task-parallelism in a natural way. However, to achieve data-parallelism, the developers have to use library functions, for operating over lists of objects in parallel, or have to manually configure all the input and output channels between the number of instances they wish to run in parallel. While the former approach is somewhat limited in its ability to express data-parallel applications, the latter implies a static configuration that might not be able to adapt to the underlying resources, but relies on assumptions about the execution environment.

As such, for any given application, partitioning the application into a pure task or data-parallel model, i.e., limiting it to either expressing the parallel execution of multiple distinct tasks or a singular task executed in parallel over subsets of a dataset, might be sufficient, however, there are a number of applications that could benefit from both types of parallelism. With multimedia algorithms being an example of one such type of applications as they operate on continuous streams of data. Limiting a parallel programming model to either direction means there are levels of parallelism that might remain unexploited. Furthermore, designing a programming model with this in mind is also preferential, otherwise one might end up with unnatural language constructs, such as with MapReduce for data parallelism. Nonetheless, having the inherent ability to model both task and data parallelism is a preferential property of any parallel programming model. As long as the goal of the programming model is not to have as simple a programming model for the application domain as possible.

3.3.4 Iterative behavior

Most applications exhibit iterative behavior at some level of granularity or in some form. This makes sense, as the discretization of data is a common process used to transfer continuous models and equations from the real world into their discrete counterparts that are more readily consumed by digital computers. Given these discreet units of captured data, one or more operations are commonly applied to each piece of data, and iteration then provides a natural way of operating over the data. For interactive multimedia applications we can see this at system level, where an H.264 encoder receives a continuous stream of discreet frames to encode from some input source, as described in section 2.2.3.

At some stage in the computation an algorithm such as k-means, which we described in-depth in section 2.2.2, might be applied. Where k-means, is an example of an iterative method. Where an iterative method generates a sequence of improving approximate solutions. An im-

plementation of an iterative method, together with a termination criteria, is called an iterative algorithm. A termination criteria for an iterative algorithm may be a set number of cycles or convergence, i.e., that the sequence of refinements is finite. If an algorithm is convergent, this convergence can be achieved through a rigorous exhaustion of the sequence domain, or where this proves impractical or impossible (from either a computational or timing perspective) a heuristic approach may be utilized. Where a heuristic approaches typically sacrifices accuracy for performance, i.e., the heuristic will not return the optimal solution, but will significantly reduce the time taken to discover a solution, memory consumption or complexity. A distinguishing feature of iterative methods is then that the progress of a subsequent iteration depends on data from one or more previous iterations. In H.264, we also see an example of these feedback loops. Motion estimation, as described in section 2.2.3, relies on previously encoded frames for predicting the motion of a macroblock. As such, motion estimation depends on the output from a previously encoded frame, i.e., an earlier iteration of processing of the input stream.

Iterations can then appear within an application at different granularities and in different forms. An important property of iteration, is that what is commonly perceived as iterative behavior can also identify a parallelization opportunity. Looking at listing 3.3, we can see an example of this that uses OpenMP [48], where OpenMP is a shared memory multiprocessing library that uses compiler directives, library routines and environment variables to influence run-time behavior. The example demonstrates a sum reduction within a for-loop, but looking at line 16, we can see how easily the OpenMP library can convert this for-loop from sequential to parallel execution. Where the OpenMP compiler directive in this case signals that for each value of *i* the body of the loop can be run in parallel and that *sum* is the scalar reduction variable.

Listing 3.3: OpenMP and iterations

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main (int argc, char *argv[])
6  {
7  int    i, n;
8  float a[100], b[100], sum;
9
10 /* Some initializations */
11 n = 100;
12 for (i=0; i < n; i++)
13   a[i] = b[i] = i * 1.0;
14 sum = 0.0;
15
16 #pragma omp parallel for reduction(+:sum)
17   for (i=0; i < n; i++)
18     sum = sum + (a[i] * b[i]);
19
20 printf("   Sum = %f\n",sum);
```

```
21
22 }
```

Furthermore, these iterative opportunities exist at multiple levels within an application, as we saw in section 2.2.3 with H.264. As such, being able to exploit nested parallelism, as described in section3.3.3 is essential to achieve a maximum benefit of parallelization.

Iterations occur frequently within interactive multimedia applications, not only at multiple levels of the computation, but also at a wide variety of granularities. As iterations typically provide parallelization opportunities, it is essential that iterations are captured to ensure scalability. As such, the programming model needs to be expressive enough to be able to capture these relationships.

3.3.5 Data centric execution

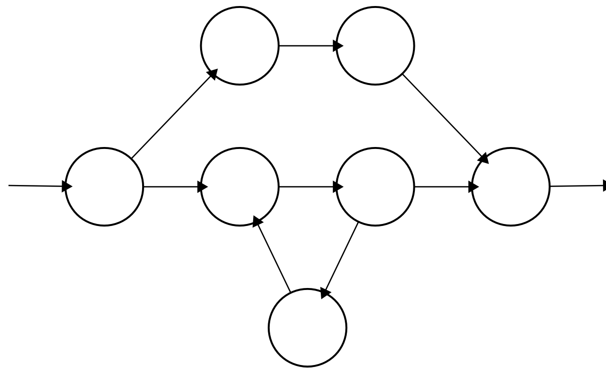


Figure 3.7: Data flow

The data flow model [49] is focused on explicitly modeling communication, where the movement of data is more important than the operations operating on the data. This is in contrast to the control flow model, implemented in languages such as C, C++, etc., where instructions are given the highest priority. A data flow programs is a directed graph, as seen in figure 3.7, where each vertex represents a function and each edge represents a signal path. The data flow model places no limitations on the structure of a program or the subsequent graph making it possible to design programs that result in arbitrary directed cyclic graphs.

The data flow principle is that any vertex can perform its computation whenever input data is available on its incoming edges. This is one of many aspects that make the data flow model similar to the functional model. The other properties are that data flow queues are immutable, there is no global state and that operations have no hidden state. The only influence one vertex has on another is the data passing through the edges. The implication of these properties is that an operation has no side effects, which is a primary property of functional languages. It also implies that the operations are always ready to be performed, at any time, no matter how many instances, the only dependency is on whether input is available on all edges. As such, the concepts of the data flow model are relatively easy to grasp, which gives us design time scalability.

A primary purpose for defining the data flow model was to provide an intuitive and easy way of using parallel resources and he data flow model inherently provides pipeline parallelism, as discussed in section 3.3.3. This is possible, because we can view each vertex in the graph

as a separate thread of execution that runs when all the data on its inputs are available. The implication being that many vertices then can fire simultaneously as long as their input data is available and there are sufficient resources available.

Kahn Process Networks (KPNs) [44] are based on the data flow programming model. Where a group of deterministic sequential processes communicate through unbounded FIFO channels, firing when the data on their inputs are available. In KPNS, the vertices block on reads when there is no data available, this results in a process network that exhibits deterministic behavior, and thus does not depend on the various computation or communication delays that can exist in the network. This also makes them particularly useful for distributed systems. While determinism is not part of the data flow model it is a property that provides a number of benefits, as described in section refsec:determinism.

Actor model The actor model is closely related to data flow, with Erlang being a primary example of its success. Unlike KPNs, Erlang does not provide deterministic execution.

Synchronous Data Flow Synchronous data flow programs [50] are a special case of data flow programs, where the number of data samples produced or consumed by each node on each invocation is specified statically. This then makes it possible to schedule a vertex, i.e., function, during compile time onto single or parallel programmable processors so the run-time overhead usually associated with data flow evaporates. StreamIT [7] applies this approach with static scheduling. However, a side-effect of static scheduling is that it is not possible to dynamically accommodate for a poor schedule or the addition or removal of resources.

Composition A compositional programming system is one in which properties of program components are preserved when those components are composed in parallel with other program components [51]. That is, the behavior of the whole is a logical combination of the behavior of the parts. A nice feature of the data flow model is that it makes it easy to create compositional programs, because fundamentally a vertex can itself be a composed graph. The use of composition can greatly simplify program development by allowing program components to be developed and tested in isolation and then reused in any environment.

3.3.6 Functional programming model

The functional programming model expresses algorithms at a higher level of abstraction then their declarative counterparts, thereby substantially simplifying the task of programming, and increasing the programmer's productivity. Abstraction, expressiveness and a clear semantic model lead to concise programs which can be developed in a short time, as well as analyzed or optimized with powerful formal methods.

Furthermore, concepts from declarative programming languages and functional programming languages in particular have proven to be particularly useful for automatically or intuitively achieving parallel execution of an application. The core contribution to this fact is the concept of pure functions, which is a function that is referentially transparent and has no side-effects. Where referential transparency means that an expression can be exchanged with its value without changing the behavior of the program, i.e., the evaluation of the expression will always produce the same output given the same input. The implication of referential transparency is then that the evaluation of an expression is deterministic. Side-effect free execution

is achieved by the absence of named state, i.e., the evaluation of an expression does not change any global state or hidden local state. A pure function is therefore deterministic, stateless and without side-effects, and can therefore be applied in parallel without having to be concerned with managing the effects of concurrency.

However, research in the past years have shown that parallel programming cannot be completely hidden from the programmer, i.e., it is not possible to automatically transform an arbitrary program into a parallel program. However, functional languages more readily lend themselves to parallel execution, which is most easily achieved through declarative concurrency that ensures deterministic concurrency. It extends, the side-effect free model with threads and data flow variables. Where a thread defines a sequence of instructions, executed independently of other threads, and a data flow variable is a single-assignment variable that is used for synchronization, i.e., the data flow variable makes it possible to mimic the properties of the data flow model, as discussed in section 3.3.5. Using these primitives, a thread that will evaluate a function waits until each argument of the function is available before executing, where each argument for that thread of execution is bound to a data flow variable, which ensures that an argument is never received more then once, and thus the thread is never executed more then once.

A number of functional programming languages have been extended with support for parallel execution, where the pure functional language Haskell [52] is a notable example of this. With Nepal [53], Haskell has support for nested data parallelism, primarily through the introduction of a parallel array type and a set of parallel array operations. In addition, Haskell has a concept of pure parallelism with the *par* and *pseq* extensions [6], where the *par* construct indicates that its arguments may be evaluated in parallel in any order, and *pseq* indicates that its arguments may be evaluated in parallel, but with some ordering restrictions.

From a developer's perspective the functional model has a number of nice properties, particularly with regard to deterministic parallel execution and simple abstractions. However, the application domain of interactive multimedia algorithms still requires global state, to some extent, as the workloads commonly contain algorithms that are more readily expressed in such a format.

Summary

Feature	Erlang	Cilk	MapReduce	Dryad	StreamIt	Parallel Haskell	C++
Parallel determinism	No	No	No	No	No	Yes	No
Data parallelism	Explicit	Yes	Yes	Yes	Yes	Yes	No
Task parallelism	Yes	Yes	No	Yes	Yes	Yes	No
Nested parallelism	Yes *	No	No	Yes	No	Yes	No
Pipeline parallelism	Yes	Yes	No	No	Yes	Yes	No
Composition	No	No	No	No	No	No	No
Graph model	DCG	DAG	No	DAG	DCG	DCG	No
Data flow	Yes	Yes	No	Yes	Yes	Yes	No
Named state	Inbox	No	Yes	Yes	Yes	No	Yes

Table 3.1: Programming language concept support
DCG = *Directed cyclic graph*, DAG = *Directed acyclic graph*

There exists a great variety of programming models and languages [36], all of which are designed to solve certain sets of problems within certain application domains. For the purpose

of parallelizing interactive multimedia workloads, we perceive that there is no existing model that provides a seamless fit. As such, formulating a new model, with a basis in the discussions of the concepts mentioned thus far is a viable approach for achieving this.

There are multiple examples of the success of integrating concepts from multiple programming models, such as Cilk [54] and Cilk++ [55] that extend C and C++ with functional, continuation style passing to achieve task and data parallelism in a procedural and object-oriented language with great success. Similarly, Google's MapReduce [13] borrows the functional concepts of *map* and *reduce* to create a data-parallel processing framework, for efficient distributed processing of large scale data sets.

Looking at table 3.1, we see an overview of existing languages and their relationship to the concepts we have discussed so far. Where we can more easily see what languages can provide the required inspiration. Where, for multimedia algorithms, named state provides an intuitive way of modeling the complex inter-dependencies between functions and data. And merging such a concept with the functional concurrency model, which is heavily based on the data flow model, can help use achieve an intuitive way for the developer to write parallel applications without having to consider low-level synchronization primitives.

Given that we are able to identify the concepts we wish to integrate into a new model for parallel application development, we have to find a way of expressing this model, which is a discussion we will cover in the remainder of this chapter.

3.4 Delivery

Given a set of concepts that one would like to merge into a single parallel programming model, there are considerations that need to be made as to how one wishes to facilitate the developer in writing parallel applications. The power-law distribution [56], which is also known as the 20:80 distribution, describes how the probability of measuring a particular value of some quantity varies inversely as a power of that value. This distribution extends to the relationship between developers and their skill sets. Where 20 percent of the developers (the experts) have 80 percent of the skills (thereby the name 20:80). Given such a distribution of developers' skill sets, it is beneficial to target the majority of developers with a programming model, i.e., the 40 percent in the middle, by providing abstractions that can conceal or defer tasks that are complex and error-prone to compile time or run-time systems. However, these abstractions should not present too simplified a view of the world, otherwise one might be sacrificing performance for simplicity. Striking the balance between usability and efficiency in terms of development cost and performance is therefore necessary.

Following this trade-off between ease of use and performance, integration of parallel execution can exists anywhere in the specter from implicit to explicit parallelization. Where at the extreme end of explicit and implicit parallelization we find, respectively, manual thread creation and handling of critical regions with synchronization primitives and automatically parallelizing compilers and parallel languages. Finding the point that provides a good balance can be difficult, and there exists a number of approaches to solving this issue. Where one can either write a completely new programming language, implement a library that integrates with the existing programming language or write a coordination language. Starting with a brief discussion on automatic parallelization we will then discuss these other approaches.

3.4.1 Automatic parallelization

Being able to write sequential code and have a compiler automatically extract the inherent parallelism is the holy grail of parallel and distributed computing. There have been attempts at achieving this, using among other Fortran as a starting point, due to its strict semantics. However, the attempts have met with limited success [57]. Some of the main complicating issues are related to pointers and recursions, loops with an unknown number of iterations and issues when synchronizing access to global resources. Some of the most promising results stem from functional programming languages [58]. A main problem exists in the trade-off between writing simple programs and achieving high performance. The general consensus is that the less control the developer has over data and task placement, partitioning and granularity, the more likely the application will achieve limited speed-ups compared to their sequential execution. As such, there has been little success within this area so far, where some even indicate that it is beyond the reach of current research and that new models and hardware architectures need to lead the way [59].

The implication is that for the foreseeable future we have to provide developers with sufficient tools for providing accurate hints as to how parallelization can be achieved explicitly.

3.4.2 Library

The library approach is used by Dryad [14], MapReduce [13] and Cilk [54]. Where Dryad and MapReduce provide abstract interfaces that need to implemented by the developer. In the C++ version of Dryad, a computational vertex is derived from the base type *Vertex* that has a set of pure virtual functions that need to be implemented by the developer, and the communication patterns between the computational vertices is specified using overloaded operators. In MapReduce, the developer specifies the implementation of the *Map* and *Reduce* functions, as well as a *splitter*, which specifies how the input data should be divided between map instances. Cilk, on the other hand provides a limited set of keywords or macros that are used to annotate the code with parallel execution hints. A benefit of the Cilk approach is that any program annotated with these keywords is *serially semantic*, i.e., any Cilk program can devolve from parallel to serial execution. The trade-off with Cilk is that one can't express all types of parallel problems, such as producer and consumer, message passing, etc.

While the library approach is non-invasive and can be quite elegant, it is most useful in scenarios where the scope is limited and run-time optimizations are sufficient for achieving speed-ups. MapReduce is a trivial model for achieving data parallelism, and Cilk can even devolve to sequential execution. In the cases where trading generality for simplicity makes sense, the library approach is highly beneficial, as evidenced by both MapReduce and Cilk, which are widely used and successful within their application domains. Explicit languages have the added benefit that they can do static or compile time optimizations in addition to run-time optimizations.

3.4.3 New programming language

An approach to implementing a parallel programming model is through the definition of a completely new programming language, or extending an existing language with parallel capabilities. For pure functional languages, extending the language with parallel execution is achieved relatively easily, as demonstrated by Parallel Haskell [6]. Though as we discussed in section3.3,

there are a number of parallel programming model concepts that are beneficial to have in a language designed for interactive multimedia applications that is in stark contrast to the core concepts of functional languages. On the other hand, designing a language from scratch provides considerably more overhead, particularly when considering that a compiler, with optimizations, needs to be built from scratch. Also, maturing a language would take time, and building a user base could take considerable time with no guarantee of success, consider the Go programming language [60], developed by Google, which despite the considerable market share of Google has been unable to make it a household name.

Attempting to roll out a new language is clearly difficult to achieve, and given those circumstances, ensuring adoption of the new programming model might be difficult to achieve.

3.4.4 Coordination language

A final approach is to design a special purpose coordination language, based on the principals found in Skywriting [61] or Linda [10]. From the perspective of a coordination language a program consists of computation and coordination and there should therefore exists a clear separation between these components. As such, a coordination language can be considered orthogonal to the computation language, which makes it possible to easily interchange the computational language if required. A coordination language then facilitates a generalized view of enabling parallelism by supporting the communication and distribution of data at any level of granularity and arbitrary nesting of such communication in either distributed, heterogeneous or time-coordinated systems.

A coordination language then provides language support for expressing the coordination and is less concerned with what is being computed. Coordination is accomplished via a shared data space, where the communication is generative, i.e., agents communicate by generating data in the shared space. This data is then available to any other agent that has access to the space, this contrasts with the message-passing paradigm where communication is usually a private act between the participating agents. A coordination language then provides a powerful way of separating the distribution and synchronization of data, which are uniform operations on primarily passive data, away from the application code. Furthermore, coordination languages do not inherently guarantee deterministic access to their data.

Commonly these coordination languages include functionality for read, write, test and remove [62], typically on an associative data structure. These semantics are not necessarily well-suited for the class of applications related to multimedia algorithms. Where a multimedia algorithm typically interacts with multi-dimensional vector data, requesting data based on indices's. Furthermore, with continuous data streams, as implied by multimedia algorithms, one would like to adhere to some of the principals found in functional programming, i.e., a side-effect free shared data space, which would imply a single-assignment data store, which would facilitate deterministic execution of tasks, which is otherwise not guaranteed by a coordination language.

3.5 Conclusion

Interactive multimedia algorithms have a number of unique requirements that make existing models for parallel execution a poor match for the workloads and algorithms associated with that application domain. Determining how one can facilitate the execution and implementation of such workloads through the definition of a new programming model is therefore a primary

focus of this thesis. We have evaluated a number of key concepts and qualities that can impact the formulation of such a parallel programming model in this chapter.

Primarily, we have seen that named state is an important facet and that combining this with key concepts from functional languages and the data flow model can provide the basis for a powerful parallel programming model. Primarily, side-effect free functions, referential transparency and write-once semantics are essential for achieving the desired effect.

We have also made it clear that we want to conceal a number of error-prone and repetitive tasks from the developer, such as coordination and synchronization of data. The primary feature of the parallel programming model should be to facilitate the developer in describing what computations to execute on what data. As such, following the declarative paradigm the developer should focus on partitioning the application into parts that can be executed independently and then using a natural language for describing how each independent component should transform the data. We also want our application to perform.

To achieve these goals, we have to defer a number of important tasks to run-time. A number of the systems we have described so far have a run-time system that performs a number of key tasks on behalf of the developer. In the following chapter we will take a closer look at the requirements placed on such a run-time system, given the requirements defined in this chapter.

4 | Parallel run-time system

Run-time systems have become important facets in modern computer applications, providing critical services that are otherwise missing or lacking in the operating system. There are some arguments that run-time systems should not exist, but that the work accomplished by a run-time should be left to the OS [63]. While there is merit to this argument, the current situation is such that operating systems do not provide all the services required by modern applications, and run-time systems therefore are important.

There are a number of benefits with using a run-time system, such as their ability to defer making essential decisions to run-time. Being able to dynamically repartition tasks or change the granularity of data parallelism during run-time, due to changes in CPU loads, can be highly beneficial. Consider StreamIT [7], that statically analyses code and requires compile-time knowledge of the hardware topology and detailed information about the generated input and output of tasks to generate sets of tasks. Not only does StreamIT rely on a significant amount of information about the system and workload during compile-time, its ability to accommodate for load imbalances, shifts in user populations, etc., is very limited, which is readily achieved by a run-time system. Run-time systems also play an important part in simplifying the application development process, by freeing the developer from implementing mundane, but complex and time consuming services. The services provided by a run-time system can range in their complexity and granularity, from simple type checking or debugging to just-in-time compiling, task scheduling, garbage collection, sand boxing, data dissemination and synchronization or task and data parallelization. There is no specific set of features a run-time system is expected to implement and neither is there any expectation as to the generality of its applicability.

However, the generality of a run-time system can have an impact on the high-level programming model and its efficiency. The Java Virtual Machine (JVM) [64], for example, supports the Java programming model and implements its language syntax. The JVM has support for just-in-time compiling, garbage collection, type checking, communication primitives and much more, and is thus a very general run-time system. Google's MapReduce [13] on the other hand is a much more specific programming model, as it only supports a limited range of coarse-grained data-parallel applications, and the open-source implementation of MapReduce, Hadoop [65], is actually implemented in Java and therefore uses the JVM as its underlying run-time system. Clojure [66] is a functional programming language that compiles to JVM byte code. Depending on the generality or specificity of the services provided by the run-time system it is therefore possible to support a wide or limited range of application domains and programming models. Yet on the other hand, if the run-time system supports a more general model the more likely that the cognitive load placed on the developer is increased. Implementing parallel execution in Java is not trivial, but achieving data-parallelism in Hadoop or task-parallelism in Clojure is considerably easier. The advantage with Hadoop and Clojure is that their programming models

lift the abstractions up to a level that makes it easier for a developer to work with. However, the less control a developer has over the available resources, the less likely they are able to fully exploit them. It is therefore reasonable to assume that a parallel application written in Clojure will be slower then optimized Java code.

An important aspect of this thesis is therefore to determine how parallel or distributed execution of applications can be achieved efficiently in a run-time system, while minimizing the cognitive load on the developer. The overarching goal is then to automate or create sufficient abstractions for common activities related to parallel execution, such as task creation, data dissemination, scheduling, dispatching, etc., and to find a suitable abstraction level. The focus in this chapter will be on how the programming model relates to the run-time system and how to efficiently maintain and schedule tasks to ensure efficient execution of workloads given the resources available at a single node. To this respect, we will consider how the run-time system can provide core-functionality to the high-level programming model and how the run-time can efficiently implement essential services.

4.1 Relation to high-level programming model

In chapter 3, we discussed aspects concerning the parallel programming model. In this section we will discuss some of the aspects surrounding the relationship between the programming model and the run-time system. These are more esoteric concerns, while in the following section we will be discussing more direct problems related to important tasks that need to be solved by the run-time itself.

4.1.1 Decoupling

Decoupling is used to identify the separation of components that should not depend on each other. In our context we are interested in decoupling the high-level programming model and the run-time system. This is preferential because it makes it possible to interchange run-time systems according to application requirements and hardware constellations. In addition, if there is a tight coupling between the application programmer interface (API) of the run-time system and programming model, it is also difficult to make changes to the underlying structure.

Microsoft's Dryad [14], a C++-based data-parallel run-time system, is an example of a tightly coupled system. In Dryad, the integration between the C++ programming model and Dryad run-time system is achieved through a combination of pure virtual interfaces, inheritance and operator-overloading. The primary concern of Dryad is to facilitate the developer in defining their application as a directed acyclic graph using C++ language constructs to achieve this; where the vertices in the graph form the computation and the edges represent communication. Extensive use of operator overloading is used to achieve integration with the run-time system.

Later Microsoft developed the DryadLINQ language [67]. There were multiple reasons for this, primarily to make it possible to define iterative algorithms and to simplify the interaction with Dryad from a developer's perspective. In addition to this, DryadLINQ also decouples the programming model from the run-time system as a side-effect, which is achieved through the use of code generation.

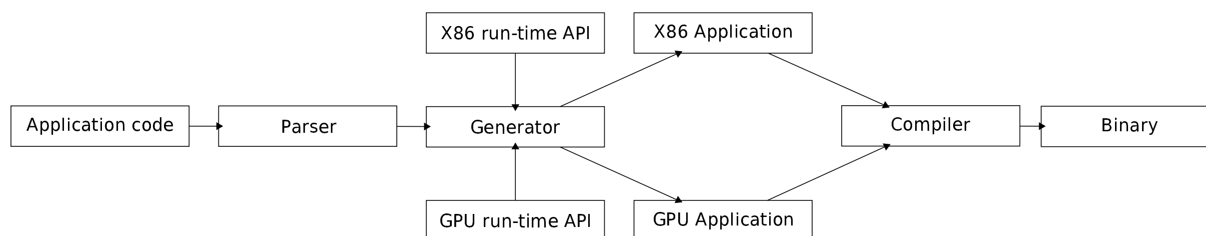


Figure 4.1: Code generation

Code generation

There has always existed a wide variety of heterogeneous hardware architectures, designed to face different challenges. However, despite the widespread existence of heterogeneous hardware these were frequently used homogeneously for performing computations. This has changed in recent years and it is now increasingly common to exploit a mix of heterogeneous hardware for the purpose of solving computationally intensive tasks. The Tianhe-1A supercomputer [68] for example, consists of 7168 Nvidia M2050 GPUs and 14336 Intel Xeon X5670 6-core CPUs, and even uses a proprietary FeiTeng-1000 CPU to enhance their Arch high-speed interconnect.

As the use of mixed heterogeneous hardware architectures is becoming increasingly common, one can decouple the application, specified in a high-level parallel programming model, from the run-time system of a given architecture, which can be achieved through code generation. The application parallelism can then be expressed at a suitable level of abstraction and appropriate code can be generated for utilizing the API of the run-time system. The run-time system can be optimized for a specific architecture, such as X86, GPU, or for exploiting certain capabilities, such as uniform or non-uniform memory access.

Manually writing multiple instances of an application, to accommodate for the potential variations in the run-time systems APIs would be a time consuming task. Code-generation then provides an efficient way of utilizing the APIs provided by respective run-time systems.

Code-generation also makes it possible to achieve more complex behavior from a run-time system then it might have been designed for. Consider MapReduce, which is limited to two stages of computation, using a code-generator multiple instances of MapReduce can be executed achieving more complex pipe-lines, or providing the possibility of supporting simple iterative algorithms. In addition, this approach means that the application developer can work in a high-level language, without any requirement of having full knowledge of the run-time system(s) that will execute the application.

This is comparable to the Java approach, where an application written in Java is compiled into byte-code that is executed by a Java Virtual Machine (JVM). The JVM has implementations for a range of operating systems. Even native code can potentially be compiled, depending on the optimizations performed by the Just-in-time (JIT) compiler. The developer is thereby presented with a unified view of all the different hardware topologies and operating systems and does not have to consider any OS or hardware specific optimizations.

Generating sequential language extensions

Extending sequential languages with parallelization constructs is a commonly used pattern. This makes it possible to reuse the benefits of a widely supported existing language and simul-

taneously achieve parallelization of code in a simplified manner.

The X10 [69] parallel programming language extends the Java programming language with parallel language constructs. An application written with X10 is parsed and processed by an analysis stage, which determines if correct X10 syntax has been used. If the code is parsed successfully code is generated and emitted that integrates with the run-time system implemented in the Java language. X10 can then reuse the advantages of a managed language such as Java, with garbage collection, type checking, etc., while achieving efficient parallelization of an application. The run-time system that runs on top of Java then handles all the details with regards to X10 specific functionality, such as places, regions and distribution. CORBA [70] also uses code generation, through its interface definition language (IDL). Shared objects are defined in the IDL and stubs and skeletons that interact with a given object request broker (ORB) are generated as necessary. CC++ [71] provides a set of keywords that extend C++ that make it possible to perform remote method invocations (RMI) and synchronize access to globally shared data-structures. Access to the shared-state is synchronized through a write-once read-many data structure. If a read is attempted before data has been written the read will block until data becomes available. Global pointers to these shared data structure ensure that anyone can read and write data to shared data structures and communication is achieved through the use of RMI. OpenMP [48] uses simple pragmas to extend sequential languages with the ability to parallelize loops.

Summary

A code-generator decouples the programming language from the run-time system, which makes it possible to more easily exchange the run-time system or use multiple run-time systems, developed for different architectures. With the advent of heterogeneous computing, this is a major benefit. Parallelism is then expressed in a high-level language using parallel constructs in some parallel programming model. This high-level code is then expanded to include calls to the appropriate run-time system, when required. Code-generation can also be used to extend an existing language with parallel constructs.

4.1.2 Representation (processing graphs)

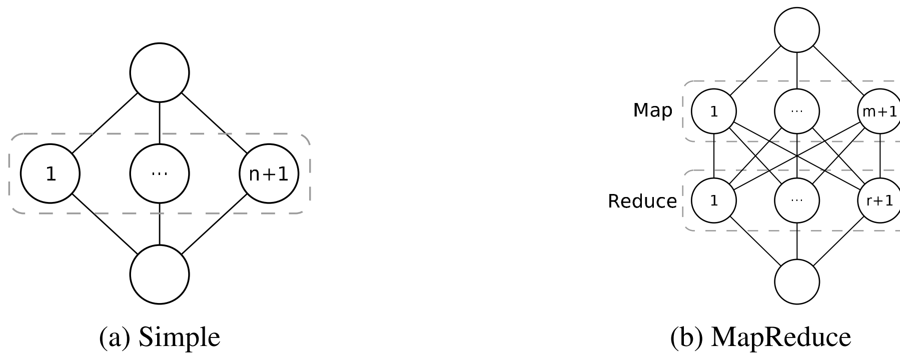
An application consists of computation and communication, where the programming model provides abstractions for specifying the interaction between these components. Following this view, we consider state as communication, i.e., the transition of a component from one state to another can be modeled as a self-loop where an edge connects a vertex to itself. For a run-time system, it is essential to make the correct decisions regarding scheduling of tasks, distribution of data and partitioning and reduction of tasks and data. Given a high-level view of the computation and communication, making the correct decisions becomes increasingly easier. In this thesis we are primarily concerned with parallel execution of tasks, and the common approach to describing interaction patterns within this domain is through the explicit specification of data flows.

By explicitly defining how the data flows through the computation, it is possible to determine during run-time where data should be sent and optimize for this, for instance by exploiting data locality and placing dependent tasks close to the data that is being produced. This makes

particular sense when we consider distributed systems, where network latencies add dimensions of time to otherwise time sensitive operations.

There are some differences in how the data flows are modeled, however, depending on the requirements of the application. The graph representation is nonetheless is a commonly applied abstraction, where the edges of the graph represent communication and the vertices represent computation.

Restricted stages



(a) Simple (b) MapReduce

Figure 4.2: Restricted stages

This is the most basic form of processing graph, and typically consists of one or more computational stages, where the simplest example of this functionality is provided by the OpenMP library and its parallelization of for-loops. The developer uses a simple macro directive to indicate that the body of a loop can be unrolled and executed in parallel over the data set it operates on. The parallelized loop represents a self-contained, single-stage parallelization. Naturally, an application can consist of a number of such isolated parallelizations of the code, but there are no explicit connections between those stages and the run-time system does not optimize for interaction between the stages. In this scenario, the run-time system is limited to local resources and also has minimal knowledge of interaction between the different components. There are therefore a limited number of optimizations that can be performed. From the run-time system perspective the primary focus is therefore to ensure that each core is fully utilized at any point during the computation. If there are four cores that can perform work at any given time and there are 100 jobs available for execution no core should be idling.

MapReduce [13] is a coarse-grained data-parallel model, where the developer is restricted to the two computational stages Map and Reduce; whose behavior is provided by the developer. There can be m Map and r Reduce operations executing in parallel, though there is a barrier between the Map and Reduce stages and it is therefore not possible for the Reduce stage to start before the Map stage is completed. The run-time system is then responsible for data distribution, coordination, etc. Due to the two-staged model, where the reduce stage depends on the output of the map stage, the run-time system can optimize the placement of data in relation to the different reducers. Designed to operate in clusters of thousands of nodes and over extremely large datasets, it is vital that data locality is used when determining where the r Reducers are placed.

Due to its simple processing pipe-line, it is relatively simple to port MapReduce to most architectures, as such, there exists a number of implementations, in addition to Google's own,

undisclosed implementation. The most well-established of these is the Apache driven open-source Hadoop implementation [65]. In addition there exists an implementation for the GPU [72], the Cell BE architecture [73] and multi-core architectures [74, 75].

In the simplest case, the primary task of the run-time system is to ensure that the processors are fully utilized and that no core remains idle as long as there is data to process. Once the processing graph becomes more complex, it is possible to do further optimizations; in the data-parallel situation optimizing for the execution of subsequent tasks based on data locality can greatly speed up the application, for instance. Determining how many

Directed acyclic graphs

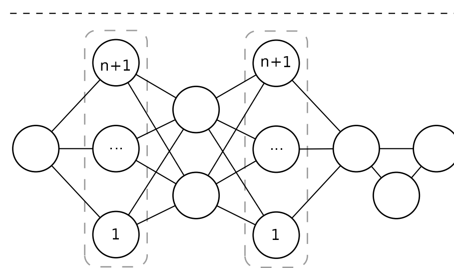


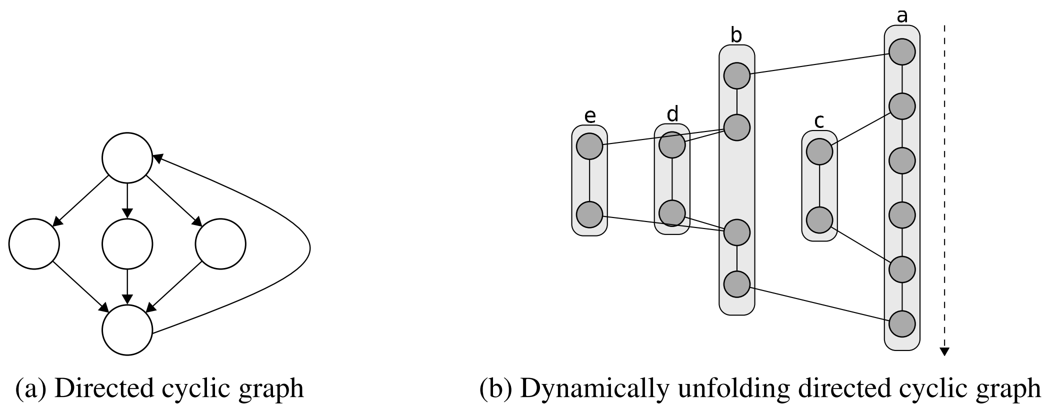
Figure 4.3: Directed acyclic graph

Limiting an application to a predetermined number of computational stages is too restrictive for a number of workloads. To this respect there are a number of programming models that make it possible to form directed acyclic graphs (DAGs). This means that the scheduling decisions become more complex, with the run-time system having to consider both interaction patterns between tasks and utilization when making scheduling decisions, i.e., a poor scheduling decision could lead to under utilization because tasks are blocked on missing input, for example.

Dryad [14] is a general purpose run-time system for data-parallel applications, where the application is modeled as a directed acyclic graph (DAG) with vertices and edges representing computation and communication respectively. A vertex can have an arbitrary number of input and output channels, which are implemented as blocking calls, as such a vertex is not executed until all the input channels contain data to read. A vertex is a simple sequential program, containing no thread creation code or locking of shared resources. A channel can be implemented in a number of ways, as shared-memory, FIFO queue, file, etc., but as its implementation derives from a global base class the interface for reading or writing data is uniform across all implementations. Parallel execution in Dryad is achieved by scheduling vertices to run simultaneously on multiple cores or nodes. Due to the semantics, it is not required that the developer has knowledge of concurrency mechanisms, such as threads and fine-grained concurrency control. Dryad deals with many of the hardest distributed computing problems, most notably resource allocation and scheduling. The Dryad run-time system also makes it possible to perform run-time graph refinement, for example, if a computation is associative and commutative, and performs a data reduction, then it can benefit from an aggregation tree. An additional layer of vertices are then inserted dynamically into the tree and local refinements are performed, before the data is pushed out onto the network. The traffic on the network can then be greatly reduced as a side-effect.

In the simple case of an acyclic graph the run-time system has a complete view of the system. It is then possible to perform run-time optimizations, such as adding additional stages to the computational graph to minimize the size of the data being communicated. It could also be possible to combine stages if it would help reduce the amount of communication. Nonetheless, as the processing graphs become more complex the run-time system is capable of performing more complex optimizations also. However, the implication is also that finding the optimal solutions means searching in a larger set of possibilities. Finding that optimal solution also requires resources; balancing between finding an optimal solution, and delivering that solution within a reasonable time then becomes important.

Directed cyclic graphs



(a) Directed cyclic graph (b) Dynamically unfolding directed cyclic graph

Figure 4.4: Directed cyclic graphs

A number of workloads perform iterative refinement of data, such as the *k*-means workload described in section 2.2.2. For these workloads, being able to express cycles as part of the programming model is essential. A run-time system that is aware of cycles can then gather instrumentation data over time in an attempt to utilize such data to refine the placement of tasks and data as the workload is being processed creating a weighted graph. Such a weighted graph could then be processed using a graph partitioning algorithm for example.

The Nornir run-time system is a Kahn Process Network (KPN) [44] implementation. A KPN has a simple representation in the form of a directed cyclic graph (DCG) with processes as vertices and channels as edges. A process encapsulates data and a single, sequential control flow that is independent of any other process. Processes are not allowed to share data and may communicate only by sending messages over channels. Writing to a channel will always succeed, but a read will block if there is no input on the channel. A process may not test a channel to determine if there is data available before reading. A channel is directed and forms only a 1:1 communication pattern. These formal properties of KPNs make them deterministic, which makes them particularly well-suited for parallel execution. Simultaneously, there is no global state, which can greatly complicate interaction patterns. Like Dryad, processes can potentially be merged or additional stages can be inserted into a computation without breaking the formalism of KPNs. KPNs are also compositional, which means that a process in a KPN network can be a KPN network in itself. Each process is then a sequential program that consumes tokens from its inputs queues and produces tokens to its output queues.

Erlang's [47] approach to concurrency is based on the Actor Model [76] of computation, which is a mathematical schema that implies that everything is an actor (much like the object-oriented philosophy of everything is an object). Actors are independent, concurrent entities that communicate by exchanging messages asynchronously. Each actor encapsulates a state and a thread of control that manipulates this state. An actor, in response to a message that it receives, can make local decisions; create more actors with a specified behavior, migrate to another computing host, send messages to actors asynchronously or alter its current state, which can possibly change its future behavior. Actors don't necessarily receive messages in the same order that they were sent, but all messages are buffered in the receiving actor's message box before being processed. Communication between actors is fair in that an actor that is infinitely often ready to receive a message will eventually get it. An actor can interact with another actor only if it has a reference to it and actor references are first class entities, i.e., actors can be passed between actors as messages to allow for arbitrary actor communication topologies. Because actors can create arbitrarily new actors, the model supports unbounded concurrency. However, because the Erlang communication unfolds dynamically, it can be difficult to perform optimizations based on a view of the graph.

Cilk [55] presents a different view of a DCG, where the graph unfolds dynamically during execution. It consists of a collection of Cilk procedures, each of which is broken into a sequence of threads, which form the vertices of the DCG. Each thread is a nonblocking C function, which means that it can run to completion without waiting or suspending once it has been invoked. A thread can spawn new threads of execution that can run concurrently with its spawner, known as successors or children, with the successors forming the edges in the graph. Since threads cannot block in the Cilk model, a thread cannot spawn children and then wait for values to be returned. Rather, the thread must additionally spawn a successor thread to receive the children's return values when they are produced. A thread and its successors or children are considered to be parts of the same Cilk procedure. Thus, a Cilk computation unfolds a spawn tree composed of procedures and the spawn edges that connect them to their children, but the execution is constrained to follow the precedence relation determined by the DAG of threads. Conceptually, the Cilk model is similar to a functional language, where the computation is described in terms of a series of function calls, with each thread or function containing no state. The Cilk model is optimized for multi-core or local execution. Due to the dynamic nature of the graph and lack of explicit communication, i.e., everything is a continuation of function calls, it is a lot more difficult to optimize the run-time execution. As such, the run-time system has to ensure that tasks are scheduled fairly according to their arrival in a run queue or similar.

The more sophisticated the graph representation becomes the better the model is able to represent different types of parallelism. To this respect, the scheduling decisions that can be made also become increasingly complex. Finding the balance between improving performance by making good decisions and overloading the CPU with logic for making the correct decisions is a difficult balance to find.

Summary

Depending on the programming model, the run-time system needs to be more or less sophisticated. The more complex graph representations make it possible to perform a number of optimizations, but frequently these optimizations also require resources. Another side-effect of a more generalized model is that it can more readily describe a broader variety of paralleliza-

tion opportunities. While this is beneficial it also means that the granularity of tasks decreases with the complexity. Creating individual threads for each task and letting the operating system schedule them becomes less and less a viable approach. We will now look at how run-time systems can support the efficient execution of tasks.

4.2 Run-time system fundamentals

A run-time system makes it possible to defer a number of important decisions, such as scheduling and dissemination of data to run-time. This is desirable, because the complexity of modern multimedia applications means that the scheduling mechanisms supplied by the OS typically are not sensitive enough to the application domain to be efficient enough. Furthermore, as run-time systems typically have a better understanding of the applications they are executing, it is possible to introduce optimizations that the OS would not be able to perform. In the following sections we will discuss how a run-time system can support the efficient execution of parallel workloads.

4.2.1 Profiling

Profiling refers to the ability of the run-time system to monitor or measure the level of a workload's performance. There are a number of ways to gather profiling data. This can be accomplished by using hardware interrupts, code instrumentation, instruction set simulation, operating system hooks and performance counters. By aggregating such information over time a run-time system can potentially perform a number of optimizations. For interactive multi-media applications this is particularly true, because these applications typically run over extended periods of time and operate on continuous streams of input data. The processing graph of a workload can then be weighted with this data during run-time, with the intent of dynamically reordering, merging or migrating tasks during execution. This data can also be stored for static evaluation, with the intent of performing compile-time optimizations retrospectively.

4.2.2 Scalable parallel execution

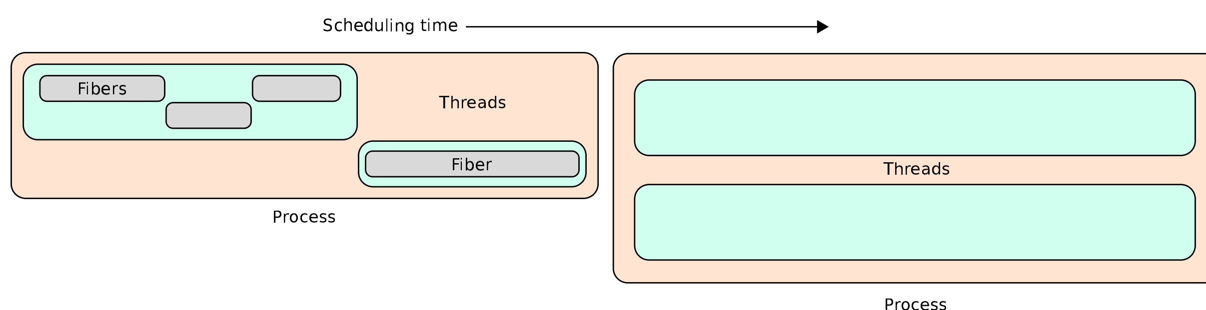


Figure 4.5: Operating system units

Parallel execution can be achieved in a number of ways using the operating system (OS) constructs of processes and threads or fibers [77,78]. An OS process is a unit of execution, with an associated exclusive address space, priority, identifier, etc., typically contained in a Process Control Block (PCB).

At any time the OS manages an arbitrary number of processes that are either user or system instantiated. These processes run concurrently on the available hardware resources and typically there are more processes than processors. A process therefore runs in a time-slice and the OS will perform a context-switch on a process when its time-slice has run to completion. A context-switch involves interrupting the executing process and storing its current state in the PCB, such as the register values of the process, accounting information (when was it last scheduled, how long has it been running, etc.) and schedule in a new process, using the information stored in the processes PCB to continue execution from the last time it was running.

These context-switches are inherently time consuming and much effort is therefore placed on minimizing the number of context-switches. Furthermore, processes do not share a common address space so all communication is achieved through inter-process communication (IPC) using sockets, signals or memory-mapped areas for instance.

Threads then provide a viable alternative to processes, as they are light-weight processes that run within a process. They mitigate the effects of expensive process context-switches and do not require IPC to share state since two threads running in the same process share the same address space. Threads have their own Thread Control Block (TCB), which is a minimal PCB that contains sufficient information to context-switch between threads running within a process. Thread context-switches are therefore considerably faster then process context-switches.

While threads can significantly speed-up an application compared to using processes and IPC, as a result of sharing memory, mutual exclusion to operations that modify shared data needs to ensured. otherwise the memory can be left in an inconsistent state. However, correctly locking access to these critical regions is a notoriously difficult task. If locks are used incorrectly one can easily create deadlocks, race conditions, or waste cycles when waiting for a lock to be released. These are not issues avoided by IPC either though, as message passing through sockets, etc, is also inherently nondeterministic.

Fibers are closely related to threads, where the main difference is that threads are preemptively scheduled, i.e., the thread's execution is interrupted by the OS, while a fiber uses cooperative scheduling, i.e., it will yield its execution to the OS that is then free to schedule in another fiber. Because fibers multi task cooperatively, thread safety is less of an issue than with preemptively scheduled threads, and synchronization constructs including spin locks and atomic operations are unnecessary when writing fibered code, as they are implicitly synchronized.

Schedulable tasks

The unit that is schedulable in a system can have a profound impact on the scalability of the system. Processes, threads and fibers are the schedulable operating system units, and they provide a fine granularity for a number of problems.

Google's MapReduce [13] for instance operates on large amounts of data across large clusters of machines. As such, using a process per mapper or reducer instance makes sense, because there are no dependencies across the data. Furthermore, while it is desirable for jobs to complete fast there is no expectation for this to be the case; the job should simply complete at some point in time. Furthermore, the operations typically operate on large datasets and subsequently large chunks of local data, the overhead introduced by process-level context-switches will therefore have minimal impact. To this respect, the benefit of cleaning up the state from the process before executing a new job might be more beneficial.

For more fine-grained applications, such as Dryad, thread-level execution might make more

sense, due to the finger granularity of tasks. However, there is a limit to the number of threads an OS can having running concurrently. Linux has a default setting of 1024 threads per process and while this number can be changed some performance degradation by the OS scheduler can be expected due to the overhead of context-switching between the threads if it is set to a too high number.

However, even threads cannot scale to the number of parallel tasks that are created by a number of parallel programming models. Erlang for instance can have thousands or even millions of concurrent tasks (called actors). Using the operating system schedulable task unit is therefore not a viable solution. Instead, having a custom wrapper around a user-level schedulable component makes more sense.

User-level light-weight tasks

Due to the overhead associated with process and thread context-switches and their large memory footprints, having large numbers of concurrently executing threads or processes for enabling the parallel execution of an application is not feasible. Modern parallel programming models enable thousands or millions of tasks to co-exist and potentially execute concurrently [47]. Consider inter coding in H.264 on frames of 1920x1080 pixels that contains 8100 macroblocks. Testing for all 8-modes, using a search window of +/- 7 pixels using one reference frame means a total of 16.5 million tasks can potentially be executed in parallel, and this is without considering variable block size, subpel motion estimation, intra coding or multiple reference frames.

Due to these characteristics, it is therefore common for modern parallel run-time systems to use light-weight tasks. Furthermore, these tasks are typically created dynamically, because there exists no knowledge of the size of the problem at compile-time and the workloads typically operate on continuous streams of data. Following our inter-coding example that means that each new frame of data can potentially result in 16.5 million new dynamically created tasks. These light-weight tasks can be implemented in a number of ways, which commonly depends on the programming model. Erlang's light-weight tasks are the actors that communicate using message passing. Erlang tasks share no state, and have their own stack and heap, and the Process Control Block (PCB) of an Erlang process is only 300 words. As such, one can have millions of Erlang processes without experiencing any performance degradation. In X10 [69], activities provide the foundation for lightweight tasks, and they can either be created locally or remotely. An activity is located within a place and never moves. A place is virtual, in that it can be moved between nodes depending on load-balancing requirements, etc.

The benefits of light-weight tasks is that context switches between two light-weight tasks in the same address space reduces the amount of state that is necessary to be saved and restored, even though saving and restoring registers is still required and still is the dominating cost of the context switch [79]. However, some frameworks have removed the need for context-switches between light-weight tasks all together. In Cilk [54], the unit of a light-weight task is a continuation and that is a C function that runs until completion, i.e., once it has been scheduled to run it can not be preempted by another continuation. A continuation is not dispatched until all dependencies, i.e., variables that it requires to execute are available. A continuation can instantiate children, but is unable to wait for a return value. If a computation depends on the return value of a child, a successor to the continuation will receive this value instead.

Light-weight tasks are also more appropriate for load-balancing in a distributed system. Migrating threads between systems is generally not an option, because threads are not necessarily

transparent. With light-weight tasks we decouple the task from the operating system specific implementation for executing code.

Furthermore, both Cilk and Erlang will only dispatch a task when all the data a task depends on is available. The implication being that only tasks that are ready to run are available in the run-queue. An operating system would not be able to know which threads were ready to run at a given time, but would have given each thread its fair share. By having a method of detecting when a task is ready to run it is possible to avoid the overhead associated with context-switching between a large number of threads.

4.2.3 Scheduling tasks

Light-weight tasks are a common way of modeling a task for achieving parallel execution in modern run-time systems. Given that this is the desirable schedulable unit, the question becomes how to handle the scheduling of these tasks and how one can optimize the utilization of the available hardware resources.

Local utilization

Given a multi-core processor and a number of tasks, the question becomes how to ensure that all the cores are fully utilized at any point in time. The common approach is to create a pool of worker threads; using one thread per core commonly. These worker threads then fetch tasks from a work-queue and process them continuously. The task of the OS is then to schedule in these worker threads, while the run-time system is responsible for ensuring that each worker thread has enough work to perform. Most modern run-time systems that operate with light-weight tasks utilize this approach, such as Erlang, Cilk, etc.

It is, however, not given that the number of threads has to match the number of cores. Activities within X10 [69] also run in a thread from a pool of worker threads, but these activities can block, and if that occurs will also block the entire thread, as activities cannot be context-switched. Therefore, if an activity blocks, a new thread is created to ensure that the application does not end up in a deadlock with all activities (threads) waiting for each other to complete. In the future, it is proposed to do as in Erlang [47], where the processes also run in worker threads, but context-switches can occur, it is then possible to store the state of the process and reschedule it at a later point in time. Another way of ensuring that a deadlock does not occur is using the Cilk [54] approach, where continuations always run until completion and cannot be blocked or preempted.

Scheduling

The run-time system scheduler then needs to ensure that each core has tasks to compute, while minimizing negative side-effects, such as false sharing (where two independent tasks read from the same page, but do not operate on the same subset of data), ping-pong effects or cache trashing (due to data moving between cores frequently), data and task or task and task locality mismatch between tightly synchronizing activities. These are a few examples of the numerous sources of overhead that may arise if tasks and data are not distributed properly by the scheduler. However, to avoid a number of these issues, the scheduler might require accurate information, both about the hardware topology and processing graph of the workload, which may not be readily available. Furthermore, calculating how to accurately place tasks in relation

to each other might require more computational resources than it saves. As a result, the work-stealing approach has proven to be a highly efficient method for achieving efficient scheduling of tasks.

The Cilk run-time system implements a two-level micro and nane-scheduler approach, where there is one nano-scheduler for each core and the micro-scheduler is responsible for ensuring that each nano-scheduler has work available. The micro-scheduler uses work-stealing to ensure that the nano-schedulers always have work waiting in their queues. The nano-scheduler is responsible for scheduling threads within a single processor and simply translates each Cilk procedure into a C function call. To enable interaction with the micro-scheduler, the nano-scheduler keeps track of the current scheduling state through a deque of frames, where a Cilk frame is a data structure which can hold the state of a procedure, and is analogous to a C activation frame.

Erlang, uses a slightly different approach relying on a continuous load-balancing algorithm using migration to balance the load across the processors and only resorting to work-stealing if a processor has no work to perform. The maximum number of runnable processes over all schedulers is measured approximately 4 times per second. This value divided by number of schedulers is then used to trigger migration of processes from one scheduler to another scheduler. When a scheduler is about to schedule in a new process it will first check if its number of runnable processes is above the max value described above and if it is it will migrate the process to another scheduler according to the migration path set up. A process migration can also occur if a scheduler runs out of work, in which case it will steal jobs from other schedulers or there is an underloaded scheduler, in which case it will also steal jobs from heavily overloaded schedulers in its migration paths.

While the work-stealing approach, with variations, provides efficient scheduling of tasks within a run-time system, other possibilities do exist. For instance, given a graph weighted with instrumentation data, one could use graph partitioning to place tasks in relation to each other. A graph partitioning algorithm has an associated cost-function, which they attempt to minimize the impact of. This can be decreasing the cost of communication for instance. Graph partitioning is widely used in the VLSI community and is frequently used to minimize the number of pins required to connect transistors. However, the overhead of utilizing more intelligent approaches should not compromise the performance of the application. The most important aspect of a scheduler is that it should be fair, i.e., any process than can be run will eventually be run.

Distributed utilization

Achieving efficient distributed utilization of resources is significantly more difficult to achieve then local utilization, because the latency added by the network adds a dimension of time to the processing pipeline. A number of the run-time systems discussed so far in this thesis therefore operate exclusively in single-node scenarios. However, run-time systems such as Erlang, Dryad and MapReduce are designed for operating under such circumstances. The actor model utilized by Erlang is particularly well-suited for this design, because everything is modeled as messages being passed between the actors' mailbox'. Erlang, however, does not take into consideration distributed resources beyond its work-stealing algorithm. If an actor creates new actors, these are instantiated locally, and only if the migration logic or work-stealing algorithm of neighboring nodes is activated does a load-balancing occur.

Hadoop, a Java implementation of MapReduce, uses a location aware file system, the

Hadoop File System (HDFS) [80], to synchronize state between nodes in the system. A location aware file system is important, because a computation requested by an application is much more efficient if it is executed near the data it operates on. The scheduling of tasks is then based on decreasing network congestion and increasing in the overall throughput of the system. The HDFS then provides the mechanisms for migrating a computation to the location where the data, or most of it, is located. In addition, HDFS will use the location awareness when replicating data. As it will attempt to place copies of the data at distinct locations.

Hadoop comes with the provided Fair scheduler and Capacity scheduler [81, 82], but can also be extended with a custom scheduler. The Fair scheduler groups jobs into pools, with each pool being assigned a guaranteed minimum share of resources or slots. If there is available capacity in the system, this is evenly distributed across the pools of jobs. The share of each pool is then split across the jobs in a pool, when a slot needs to be assigned (with a map or reduce task), if there is any job below its minimum share, it will be scheduled in. In addition, it is possible to give higher priority to certain jobs within a pool, and thus a higher share. It is also possible to weight based on the size of the job. The Capacity scheduler organizes jobs into queues, with each queue receiving a percentage of the resources in the cluster. Within each queue the resources are scheduled in a FIFO-style. However, a job can be preempted.

Optimizations

Processing graphs are a common representation for a number of light-weight task based data flow run-time systems. This model provides an intuitive way of thinking in terms of transformation of data as it flows through a system. Another benefit of this explicit definition of the interaction between computation and communication is that it makes it possible to perform run-time optimizations.

MapReduce, Dryad and KPNs are, for example, capable of adding additional stages to their computation, that do not break the semantics of the model, but can help minimize communication or reduce the amount of bandwidth required by a workload. MapReduce does this by introducing a miniature reduce-stage, where the data is first reduced locally before being sorted based on its key-value pairs. This is also similar to what Dryad does, where additional aggregate operations can be inserted into the processing graph to reduce the information locally before it is transmitted across the network. This can lead to significant reductions in the amount of required bandwidth.

With the use of light-weight tasks it is also possible to define a very fine granularity of parallelism. Such as with the H.264 inter-coding with 16.5 million potentially parallel tasks. However, the overhead associated with defining that number of parallel tasks is most likely greater than just executing multiple of those tasks at a higher granularity sequentially. Then, being able to merge or fine-tune the granularity of the parallel nature of the application during run-time can provide a number of benefits.

4.3 Conclusion

Processing graphs as a result of data flow semantics for modeling the interaction between computation and communication provide a powerful abstraction for understanding parallel workloads. An understanding that can potentially facilitate the developer in creating complex workloads and one that is useful for a run-time system when making scheduling decisions. A process-

ing graph weighted with instrumentation data can be used for static evaluation or for run-time optimizations using graph partitioning mechanisms or similar.

Nonetheless, given a processing graph, efficiently scheduling in the parallel tasks is achieved with the least overhead when a thread per core is utilized. A work-stealing mechanism can then ensure that each core never idles as long as their are tasks available to run. Light-weight tasks are well-suited for this, because modern workloads, such as inter coding in H.264 make it possible to have thousands or even millions of concurrent tasks. The overhead of traditional operating systems mechanisms is therefore to great. Furthermore, operating system specific schedulable units are not necessarily transparent and can't be easily migrated, if distributed utilization of resources is desired.

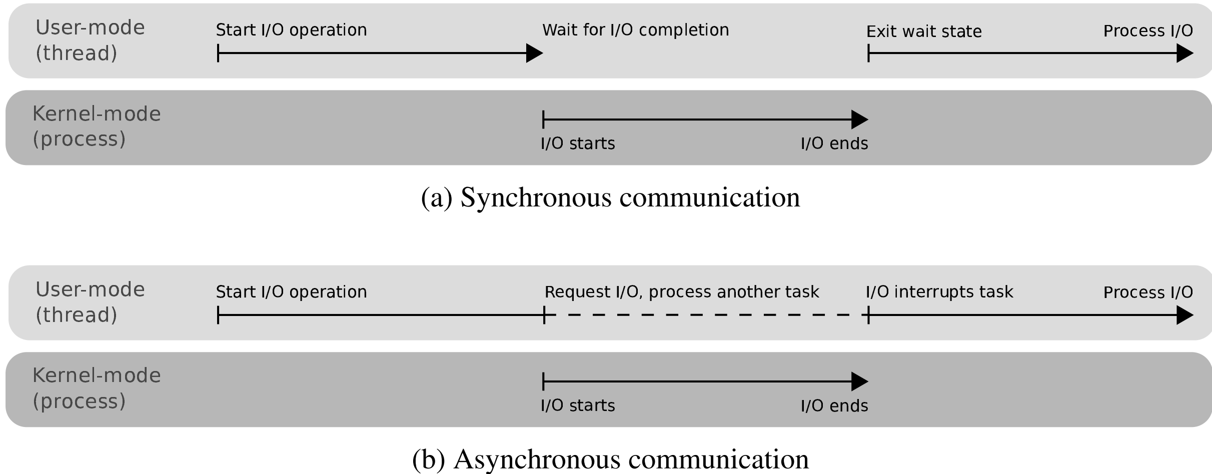
5 | Communication model

The focus of this chapter is on the subset of functionality related to synchronizing state when the state is shared among multiple nodes in a distributed system, multiple processes that do not share an address space or combinations of these. The communication model is closely related to the run-time system and provides the subset of functionality that enables the distributed processing of tasks in a topology of shared hardware resources. A primary focus of this chapter is on the introduction of communication techniques, from implicit to explicit communication, and their advantages and limitations.

Dissemination of state is required to ensure progress in workloads where tasks depend on the output of other tasks and those dependent tasks run in different address spaces. Distinct address spaces can occur for a number of reasons, because the workload is distributed across multiple processes on the same node or in a distributed system or because the memory architecture does not support cache-coherence. Synchronization of tasks is then achieved by the transmission of state, either locally or across a network. Dissemination of state can be achieved through explicit or implicit communication. Explicit communication is commonly referred to as shared-nothing communication, and implicit communication is referred to as shared-everything. A primary objective of the dissemination process is that it should be efficient, to ensure that the network and hardware resources are fully utilized, which can be achieved by overlapping computation and communication.

5.1 Communication modes

Communication can be either synchronous or asynchronous, as detailed in figure 5.1. With synchronous communication (see figure 5.1a), a thread processing a task will block until the I/O operation has completed allowing other processes to run in the meanwhile. The implication of this behavior is that we halt the progress of the workload while waiting for the I/O operation to end. This means we are potentially letting other processes use cycles that we could otherwise have used to run another task in our workload, which is not blocked pending an I/O operation. With asynchronous I/O operations (see figure 5.1b), we do not block on the I/O operation, but rather we allow for other tasks to run while we wait for the operation to complete. Asynchronous communication supports the pattern of overlapping computation and communication within a process, which is highly beneficial, particularly when we consider that the workloads we are operating on consists of large number of small concurrent tasks, such as the H.264 inter coding example, where determining how to code the individual macroblocks in a 1920x1080 frame can result in 16.5 million concurrent tasks (see section 4.2.2). If a large number of these tasks is blocking while waiting for data to become available and we are using synchronous communication a considerable time is spent waiting for data to arrive, while other tasks could executing,



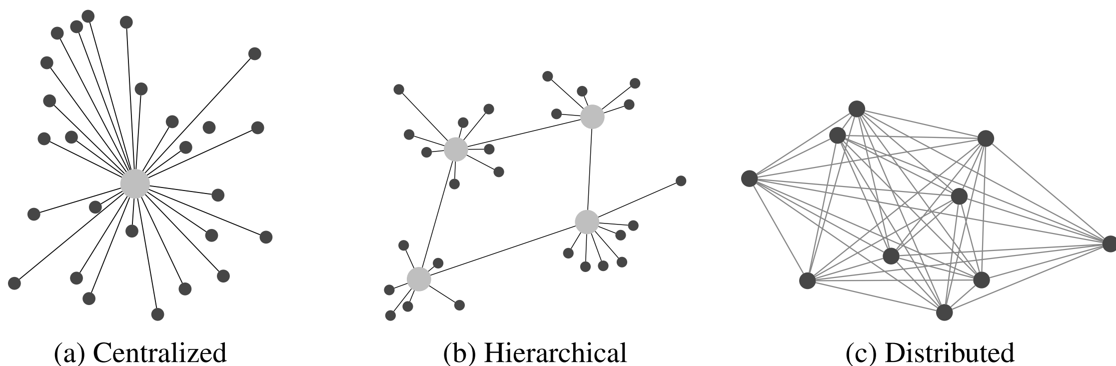
(a) Synchronous communication

(b) Asynchronous communication

Figure 5.1: Communication modes

which can greatly reduce the performance of the executing workload. Asynchronous communication is supported by user-level light-weight tasks, as discussed in section 4.2.2, where it is possible to defer the execution of a task until all data it depends on has arrived.

5.2 Name service



(a) Centralized (b) Hierarchical (c) Distributed

Figure 5.2: Name services

A name service is used to locate state, objects or services in a distributed system. Depending on the communication form, i.e., explicit or implicit, the state can be identified in a number of different ways. In an object based explicit communication system, such as Java, the object instances are commonly the units of state that are registered with the name service. In Ivy [11], a distributed shared-memory implicit communication system, the unit that is being accessed is a page of data, and as such, a virtual memory address can be the unit that is registered with the name service.

There exists a number of different ways of implementing a name service, as seen in figure 5.2. In the centralized approach (see figure5.2a), the location of all state is registered at a single location. A centralized name service can easily become a bottleneck in the system, depending on the number of registrations and queries for distributed state, though under some

circumstances this is not problematic, i.e., with JavaRMI [83] and CORBA [84] object instances are registered with the name service. However, these objects typically implement business services, which are static in their nature, and objects therefore typically persist at the same location throughout their lifetime, and furthermore, they are typically never constructed and destructed frequently (if at all). So, even though migration of objects is supported, this is not a common use-case. As such, a centralized name service, coupled with local caches of object references, can provide a scalable service.

When state is being created, destroyed, registered, queried and migrated more frequently, a centralized approach can become a bottleneck. Also, with a centralized name service, all state is treated equally, while some state might be more important to make available globally then other. Making use of an hierarchical name service (see figure 5.2b) can then provide the required flexibility, granularity and accessibility to state. The Mentat system [85] makes it possible to bind an object to be visible at a local, cluster or global level, which determines whether other Mentat objects can view that object from another part of the system. Mentat then uses a hierarchical search to determine the location of an object, i.e., first searching locally, then within its cluster and finally globally. An object that is bound locally will then only be available to other local objects.

The last approach is to use a distributed name service (see figure 5.2c), where state is registered at the location where it is created and can only be queried directly between nodes. Under circumstances where state is migrated frequently between distinct locations, this is very useful, as it removes the bottleneck of having to register with a centralized instance. Ivy [11] makes use of a distributed name service, where pages are the units of state that are registered with the name service. Each page has an associated owner, where each node in the system is responsible for keeping track of the pages associated with it. When a process attempts to access a page that is not residing in local memory, a page-fault is triggered. The requesting node will then contact the owner of that page and either request a write-access, which implies that the owner of that page will change, or a read-access, which implies that a replica of the page is requested. As such, Ivy uses replication to allow for multiple read-copies of a page. These have no associated owner, i.e., only the write-page has an owner. Since pages can migrate throughout the system, and page requests occur frequently in unstructured distributed shared memory systems, a distributed name service provides an efficient scheme for locating pages. A problem with this approach, however, is that chained-invocations might occur. This happens when a page has migrated frequently, and the requesting node needs to follow the migration path before it locates the page in question. Ivy approaches this problem by collapsing these pointer chains, by updating each name service as to the pages new location.

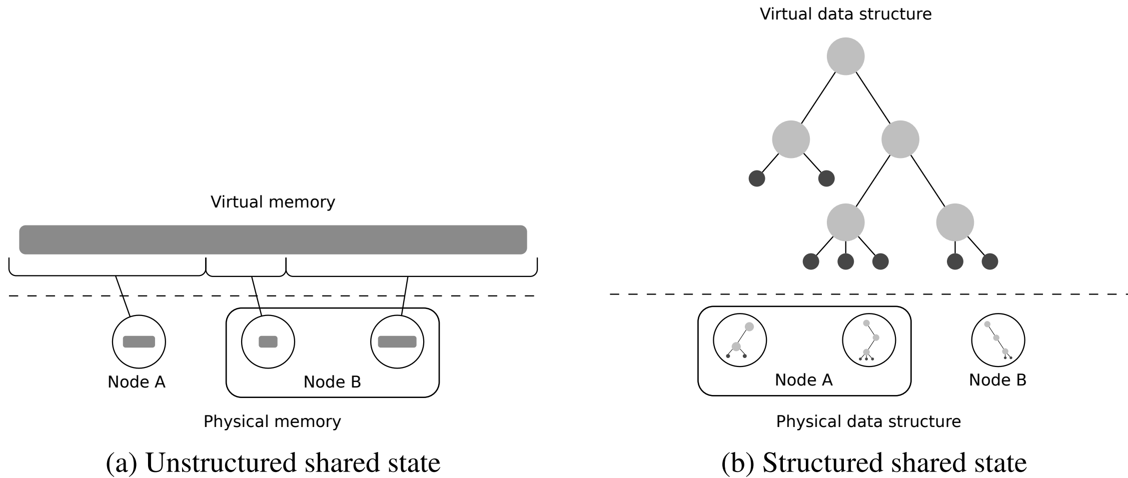
A name service is not always required, in MPI [9] it is possible to statically assign tasks to specific nodes and cores at compile-time or during start-up, providing each task with a full overview of the location of dependent tasks.

5.3 State communication semantics

How efficiently shared state is accessed or disseminated depends on the semantics of the read, write, send or receive operations. With the implications of utilizing either implicit or explicit storage semantics affecting both the system and developer, where one factor that is potentially effected is the granularity and specificity of state that is communicated. The more explicit we

are, the better control we have on what data should be going where.

5.3.1 Implicit dissemination of state



(a) Unstructured shared state (b) Structured shared state

Figure 5.3: Implicit communication models

With implicit communication or a shared-everything model, the state, which is physically distributed across multiple processes and nodes, is presented logically to the application as existing completely in local memory. Writing or reading from the logically shared state can therefore lead to implicit synchronization points, where non-local or dirty state needs to be requested and staged before being made available for use. The benefit of implicit communication is that it provides the developer with a uniform view of the shared state, where reading from and writing to the shared state is performed in the same way whether the state is local or not. Maintaining a consistent view of the state throughout all nodes and processes is deferred to a run-time system or an operating system mechanism. The primary objective of an implicit communication model is therefore to ensure that the state remains consistent according to a coherency protocol and its consistency model.

The logical representation of the shared state can be either structured or indiscriminate. The structured view of the state is commonly presented through a data structure, as seen in figure 5.3b, where Linda [10] is an example of this approach. The elements of the data structure can then be located in local memory or at a remote location. With the indiscriminate approach, the memory of the processes and nodes in the distributed system is combined into a continuous virtual memory, as seen in figure 5.3a, where the distributed-shared memory approach [11, 12] is an example of this. The indiscriminate approach is given its name because we are operating on data located within a page of memory, as such, writing to a single variable within a page of data can indiscriminately mark the entire page as dirty on all nodes in the system.

The main difference between these approaches lies in the granularity of state that is effected by read and write operations. In the indiscriminate approach, state unaffected by our write-operation has to be propagated to dependent processes, because the unit of state that we are distributing is a page of data. In the structured approach, we are reading and writing to elements of a data structure at a more accurate level.

In the indiscriminate approach, this inability to distribute only modified data can lead to false sharing, which is an inherent artifact of implicitly synchronized state protocols. False

sharing occurs when two processes operate on independent data in the same memory address region, forcing the whole page of data to be transferred across the communication network with each write, causing memory stalls in addition to wasting system bandwidth.

In either approach, maintaining a consistent view of the shared state is a primary concern; caused by the implicit dissemination of state.

Coherency protocol

A coherency protocol implements a consistency model, where there exists a broad spectrum of consistency models where sequential consistency and release consistency are the most broadly used consistency models in distributed systems using shared state [86]. Sequential consistency is provided if every node of the system sees the write operations on the same memory part (page, virtual object, element, etc.) in the same order. Release consistency is achieved by using two special synchronization operations (release and acquire) and all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter can acquire it. In this thesis, we will only provisionally discuss coherency protocols, as providing an exhaustive overview of cache coherency protocol and consistency models is beyond our scope.

The application requirements can greatly effect the coherency protocol. A primary concern is then to determine how the application interacts with the shared state, where the interaction can be primarily producer-consumer, i.e., one node is only writing data while the other is only reading it, frequent read or write or frequent read and write. This can have an impact on the coherency protocol which is responsible for maintaining a consistent view of the caches in a system of shared state. The coherency protocol implementation may choose different update and invalidation transitions, such as read-update, write-update, read-invalidate, or write-invalidate, as long as the consistency model is ensured.

A consistency model can then be achieved in a number of different ways, where a common strategy for achieving sequential consistency is achieved by using write-invalidate semantics. For indiscriminate shared state, operating at a page-level, fetching shared state is commonly triggered by a page-fault, i.e., the application is attempting to access a page of data that is currently not residing in local memory. Then, when the page-fault is triggered, the run-time system can query the name service (discussed in section 5.2) for the location of that page of data, if it is currently managed by a remote location a request for that page of data is issued. For structured shared state, the semantics are equivalent where requesting an element of the data structure that is not in local memory will trigger a request for the shared state. Within this context, read and write requests are two distinct types of requests. Where a read requests results in a read-copy of a page or element of data. A write request results in a write-copy of the page, which can then be modified by the requesting party. There is never more then one write-copy, but there can exist multiple read-copies of a page or element of data. The primary objective is then to keep the read-copies consistent with the write-copy. With write-invalidate operations, all replicas of the page are invalidated before writing to the page. Updates to a page are then only propagated when data is read, which means that several writes can be performed to the page before a reader re-requests the page.

Migration

Migration of state occurs frequently in the shared-everything model, where it is used to transfer the ownership of a write-page, i.e., a page that can only be written to by one party at the time.

Migration is also used in this context to ensure resource locality through the replication of state.

Write-once semantics

Utilizing write-once semantics can greatly simplify the implementation of a consistency model for the cache-coherency protocol. When write-once semantics are applied to the shared state, we can freely create read-copies of the shared state knowing that the data contained within the read-copy will never change. This approach is more easily applied to a structured shared state approach, but has also been implemented in indiscriminate shared state implementations, such as Munin [87]. However, the utilization of such an approach might require the implementation of a garbage collector, to ensure that stale state is cleaned up, which in turn requires mechanisms for knowing when to release the shared state.

Summary

While implicit communication provides the developer with a simplified view of the shared state of a system, it can result in the need for implementing a complex consistency model. In such a case, a mismatch between the consistency model and application requirements can lead to performance degradations due to thrashing, false sharing or other related problems. Write-once semantics can then provide a tool for simplifying the consistency model, but it also requires the implementation of a garbage collector to ensure that stale data is cleaned up.

The implicit communication model provides a trade-off between ease of programming and performance, where the implicit model generates more communication as a result of the coherency protocol that is maintaining a consistent state of the state throughout the system. This makes the implicit model less efficient than an explicit communication model, but this is considered a viable compromise for ease of use.

From an application developers perspective, implicit communication is easy to work with, since the memory appears to exist continuously in memory, and all communication is performed implicitly. A negative side-effect of this is that maintaining consistency in the memory across all the nodes requires more communication, e.g., when migrating write-pages and for invalidating replicated pages, than the explicit communication models.

5.3.2 Explicit dissemination of state

With explicit communication, or the shared-nothing model, each communicating entity or task has a separate address space, and all coordination of tasks is achieved through explicit communication channels, as seen in figure 5.4. A benefit of explicit communication is that the receiving party is known, as such, the data can be made available directly in the memory area of the receiver. There is no need to stage the data at an intermediate location before arriving at the receiver. A down-side is that it can be demanding to define the explicit communication for workloads that are irregular or have complex task inter-dependencies. Another distinguishing feature of explicit communication is that the entity that encapsulates state also encapsulates computation, i.e., the arrival of state will typically result in the invocation of a computation, whereas in the implicit model, the computation blocks if the data is not available.

There are a variety of approaches to implementing explicit communication models. The primary differentiators for these models is based on the granularity of entities encapsulating state and their corresponding send and receive semantics, e.g., if they poll for data or block

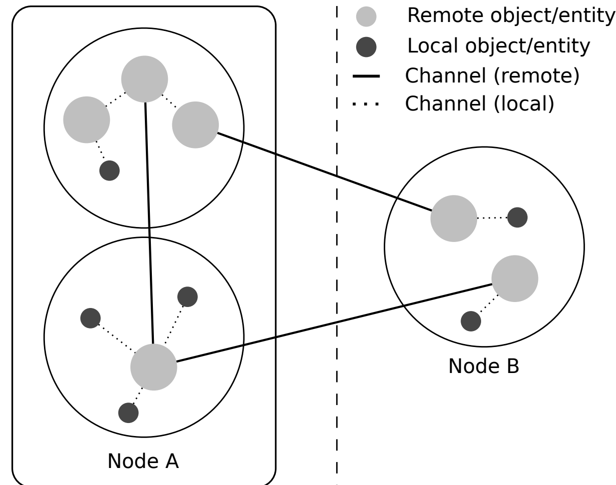


Figure 5.4: Explicit communication model

when data is unavailable. Furthermore, a primary distinguishing feature is whether the explicit communication results in immediate or deferred invocations.

Typical for the shared-nothing model is the fact that the distributed nature of the memory system is fully exposed to the application programmer. The programmer needs to keep in mind where the data is, decide when to communicate with other tasks, whom to communicate with, and what to communicate. As such, the developer is fully responsible for partitioning the code and data, and it is assumed they are capable of doing this well.

Immediate invocations

Immediate invocations occur when the encapsulating entity can process the request as soon as it arrives, i.e., all data required to perform a computation is available at the time of the arrival of the request. Depending on the implementation, the encapsulating entity can vary from processes to objects. Commonly, the granularity of the encapsulation will have an impact on the mobility of the entities, where a coarse grained entity is more likely to be static in nature. This differs from the implicit communication model, where state is frequently migrated between nodes in the system, requiring frequent updates to the name service.

Message passing uses coarse grained encapsulation, providing two key aspects, synchronization of tasks and read and write access for each processor to the memory of all other processors. Message passing is primarily push-based, as such, each process knows which processes that require its output, and ensures that this data is sent to the receiving process when ready. Message passing is derived from the early efforts of scientists and engineers developing Grand Challenge applications for Massively Parallel Processors (MPP) and is therefore well-suited for scientific computations. Oak Ridge's PVM [8] and MPI [9] are two well-known high-level message passing systems.

An RMI is similar to message passing in its approach to synchronizing state, but while message passing is designed for passing data and instructions between two or more parallel processes, RMI provides a way of calling a subroutine of an object, with the (potential) arguments passed to it representing the data (and of course a potential return value). This way, RMI can be both push and pull-based, where an invocation might result in a return value, i.e., pull-based, or just a simple invocation with arguments, i.e., push-based. While message pass-

ing most likely involves a form of method invocation, i.e., as a result of the arrival of data a method is invoked, a method invocation does not necessarily involve message passing because a method receiving no arguments might be invoked. RMI is therefore more flexible as to which application domains it supports when compared to message passing, which is typically applied to high performance computational applications.

In the object oriented paradigm, an object provides an interface of methods for interacting with the object. Through these methods other objects in the system get access to the internal state of the object. With RMI, the interface of the object is available even when objects are distributed across multiple nodes. Remote Procedure Calls (RPC) are closely related to RMI, where the main distinguishing feature is that, in RPC, one calls a function, while in RMI one invokes the method of an object instance, which requires a means of identifying what object should receive the invocation.

In the RMI-model, objects are considered abstract data types, which encapsulate their internal state through a well-defined interface. An RMI then achieves communication in a system of distributed objects. From the developers perspective, a remote and local call to an object's methods are identical. Communication is still explicit, but the location of an object is transparent.. Though, typically for these systems, it is required to provide additional information for the methods or classes that should support RMI, as we will see for our example systems.

Deferred invocations

Light-weight tasks are fine-grained entities encapsulating computation, where the invocation of the computation is deferred until all state that the entity is operating on is present, where the state can arrive at distinct points in time.

Kahn Process Networks (KPNs) [44] and Erlang [47] implement this style of invocations. In the case of KPNs, the receiving entity, called a process, has multiple unidirectional one-to-one communication channels associated with it. Dependent processes receive discreet messages on these channels, and the process cannot invoke its computational stage before it has read a piece of data from each communication channel.

In Erlang, the entity encapsulating a computation is called an actor, which has an associated mailbox. Other actors in the system send messages to this mailbox, and the receiving actor can perform its computation when the content of the mailbox matches a specific filter. At this point in time, the actor reads the content from the mailbox and performs its computation.

Deferred invocations more closely resemble the implicit communication model, but they still have explicit communication channels and have the added benefit that the computation is not started until it is certain that all the state it depends on has arrived.

Migration

With message passing, and to some extent RMI, the location of entities is typically static. The location of an entity encapsulating state is either pinned to a specific location by the developer or assigned a processor during start-up. Commonly, in message passing, the terms process and processor are interchangeable, i.e., there will typically exist one process per processor. As such, either the developer has to assign heavy processes to certain locations, or ensure that the processes are near homogeneous in their processing requirements, which again implies that the available resources have near the same capabilities. This is a reasonable assumption, because message passing has evolved from the domain of scientific computing. Similarly, objects

typically encapsulate a service, and are therefore relatively static in their nature. Objects in CORBA [70] and JavaRMI [83], for example, are generally placed at a static location and migration of objects occurs rarely, though it is feasible. Instead object discovery or service discovery is a more central issue. To a greater extent, RMI is concerned about connecting decoupled components. As such, availability, reliability and discovery of functionality is more important.

Light-weight tasks, as exemplified by Erlang's actors or KPN's processes are considerably more mobile. These frameworks are more commonly based around the concepts of computations and are designed to scale to large numbers of concurrent tasks. These light-weight tasks have no association with a particular processor or node, but are rather executed at a location where resources are available. Migration of tasks in this case is not motivated by accessibility, as it is in the implicit communication models, but rather around the exploitation of available hardware resources.

Receive semantics

The receive semantics define how the encapsulating entity interacts with the state that it is receiving on its communication channel. With KPNs, a process reading from a channel will block if there is no data available on a channel, and the progress will only continue when data becomes available. For KPNs, blocking on read forms part of the formalism of the networks and exist to ensure that the KPN is deterministic and polling for data is therefore not possible. In message passing systems, such as MPI, both blocking and polling are allowed, i.e., a process in MPI can then poll for data. Compared to message passing, RMI also specifies a method that should be run, when a message arrives, this implies that it is not necessary to poll for data or block on a read, it is rather invoked when the request arrives.

Summary The explicit communication model enables better control over the communication patterns between tasks by making the time and point of communication explicit. Typically, the migration of state in these systems implies relocating the task that depends on the data that is being disseminated. As a result, it is not common to relocate the computation, which means that the ability to adopt to changes in load or hardware resources is more limited. Furthermore, defining explicit communication channels is difficult for workloads that consist of complex and irregular interaction patterns.

5.4 Conclusion

For high-performance applications, message passing provides a viable approach given that the processes in the computation perform a reasonably similar amount of work, and that the cores performing the work are relatively homogeneous, because it is generally assumed that there is a one-to-one mapping between tasks and cores. It also assumes that the processing graph of the application is relatively simple, since all communication is expressed explicitly. This implies that the allocation of work to resources is relatively static and the communication patterns are simple. Though it is possible to add additional resources, and there is some support for migration. It is also typically assumed that the developer is capable of partitioning the functionality and data of the application in a reasonable way.

For business applications, where there are strict requirements for availability and discovery of functionality, RMI is a viable approach. Typical for these systems is then the ability to

interact with a name service, that is commonly a centralized service, to discover objects that can perform certain tasks. Here, objects are typically instantiated at one location and remain there throughout their lifetime. The frequency of object creation and destruction is also fairly limited, and with local caches of object references, the centralized name service rarely becomes a bottleneck.

For applications with irregular and complex processing graphs, the implicit communication models are viable approaches. In these applications, the developer can use simple read and writes to the shared memory, and all communication between nodes will occur implicitly. A side-effect of implicit communication is the requirement to keep a consistent state of the memory across all nodes. Since replication is frequently used to increase performance, this means that more communication is required when compared to RMI and message passing, which can incur a performance penalty. This is alleviated to some extent with techniques for annotating different parts of the memory with information regarding the type of data stored there or by applying write-once semantics. It is then possible for the run-time system to apply different consistency models to improve performance.

There also exists shared data structures that make it possible to decouple the interaction between communicating processes both in time and space. This can be useful for applications where decoupling computation is desired.

To conclude, the choice of communication model depends on the application domain. Though there are some indications of their areas of use, the message passing paradigm suites better to heterogeneous architectures as it makes no assumptions on how memory is accessed or utilized in the various systems, and discreet messages are easier to transform between various environments. Asynchronous communication is highly desirable to overlap computation with communication. The message passing paradigm is better supported for this type of functionality, though RMI can be extended with this functionality by returning a Future object. An RMI then returns immediately, but the calling thread will block on the future until it is woken up when the result arrives. It is then possible for other threads of execution to run in this space of time.

6 | Papers and contributions

In this chapter, we will provide an overview of the research contributions of each individual paper. The papers have been grouped by their prototype to provide a conceptual context. Based on these contexts, we discuss the contributions of each group and their combined thesis contributions, comparing our approaches and results to the related work presented in the previous chapters.

6.1 Overview of research papers

For the reader's convenience, we have organized the research contributions of this thesis in groups by their system implementation. The context of each group of contributions is introduced and a brief overview and motivation is given. We have sorted the groups chronologically, and they consist of *Ginnungagap* (section 6.2), *Nornir* (section 6.3) and *P2G* (section 6.4).

Within each group, we describe the main contribution of each paper and how that paper relates to the overall thesis goals. Since each paper is self-contained, some information is necessarily repeated in the different papers. For information about where the paper has been published, the evaluation process and the contributions of the authors, the reader is referred to the cover pages of each research paper, presented in Part II.

In short, Beskow has been the driving force behind papers I, II, III, IV and V. In papers VI, VII and VIII, Beskow was an auxiliary author and provided workloads for testing the system. In papers IX, X and XI, Beskow and Espeland were the primary contributors, with Beskow implementing a major part of the system code and paper contents.

6.2 Ginnungagap

Ginnungagap was developed to support the development of massively multi-player online games (MMOGs). It consists of a run-time system for managing game state and coordinating the distribution of this state in a system of distributed nodes. A C++ object model defines an object hierarchy that integrates with the application programmer interface provided by the run-time system. Extensions to the C++ syntax, with a corresponding parser and code-generator, is provided for simplifying the integration of application code with the run-time system.

Ginnungagap uses a combination of Remote Method Invocations (RMI) to synchronize state, migration to ensure resource locality and a distributed name service for efficient maintenance of references to migrated objects. As such, with Ginnungagap we merge the shared-nothing model of RMI, for efficient communication, and the migration and distributed name

service of distributed shared memory systems to ensure resource locality to improve overall performance.

Paper I: Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State

In Paper I, we explore the characteristics of MMOGs and examine the importance of latency as a measurement of a user's Quality of Experience (QoE). Based on our observations, we formulate a set of components that we consider important for supporting the development of MMOGs, with a focus on run-time system support and the introduction of a high-level programming model. No system implementation existed at this point in time, but rather an exploration of ideas was performed. At the time, MMOGs were some of the fastest growing applications within interactive multimedia applications. That, coupled with their distinct characteristics and often conflicting goals, i.e., ensure low latency for a very large number of interacting users, made it a particularly interesting topic to explore.

A key observation made in this paper is that the physical distribution of users and their virtual location within the virtual environment are distinct. The implication being that since MMOGs are intrinsically decomposable systems, the organization of these virtual regions can be decided based on user statistics during run-time and not statically during deployment. For example, game state could be migrated dynamically during run-time based on the latencies of the interacting users, thus removing the reliance on good static partitioning schemes. Given this outset, the primary contribution of this paper was to identify methods with which we could support this form of deferred decision making to run-time. Migration of game state was then an obvious feature, but less apparent was how to determine an optimal solution for maintaining references. The paper therefore explores the benefits of different approaches to reference maintenance.

We also utilized RMI as a key point to ensure message passing communication, as described in section 5.3.2, which is a very efficient method of communication in a distributed system. Furthermore, it was understood that integrating a run-time system, such as the one described, with an application would not necessarily be a straightforward task. We therefore explored the possibility of providing a high-level programming model, in which the user could define parts of their application, and which we could then parse and generate code for integration with the run-time system.

Paper II: Latency Reduction by Dynamic Core Selection and Partial Migration of Game State

In Paper II, we expanded on the concepts surrounding migration, and tested the feasibility of using a distributed name service for efficient reference maintenance in a distributed system consisting of objects, completing our first proof-of-concept implementation of the initial system implementation, which enabled us to verify the conceptual ideas first examined in Paper I.

In addition, we expand on our knowledge of service selection methodologies and consider what approaches to use for determining where to place a virtual region in relation to a set of interacting users placed logically within it. The effort of placement was motivated by the work of Knut-Helge Vik.

Due to the existence of a proof-of-concept implementation of the system, we were also able to refine the implementation details of the system, with a particular focus on how the distributed name service could be implemented. We also performed some basic tests to determine how much overhead is introduced by the existence of a run-time system.

Paper III: Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction

Paper III presents the results of our first full implementation of Ginnungagap, where Erikstad contributed to the new implementation. The major contribution of this paper is a demonstration of the feasibility of the concepts identified in the previous two papers (Paper I and Paper II).

This paper also validated the use of prototyping of real systems as we identified issues related to synchronization of state and object reference maintenance. This was done by determining the importance of minimizing the existence of chained references, as these could increase the look-up time when using a distributed name service. We also reimplemented the system for unique object identifiers by moving towards the use of universally unique identifiers (UUIDs), which remain unique for an object throughout its lifetime. This effectively removed the need to update the sending parties name service with the new object identifier. With the existence of a real system, we were also able to run extensive tests in PlanetLab, which is a distributed research network consisting of nodes located across the world. We ran several different tests, primarily to test the run-time system with its migration and distributed name service. We also ran several micro benchmarks, to test the functionality and scalability of the migration and serialization mechanisms. Erikstad also contributed with the creation and running of these tests.

Paper IV: The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency

In this journal paper, we expand on the knowledge gained in Paper II. The focus is therefore on how accurately measuring the latencies of the interacting users can impact the server selection result. Furthermore, as the server selection measurements are time consuming for large numbers of users, we explore how latency estimation can be used and to what degree the server selection result deteriorates as a result. The latency measurement and estimation techniques were results of the work of Knut-Helge Vik.

Paper V: Reducing game latency by migration, core-selection and TCP modifications

In this journal paper, we expand on the knowledge gained in Paper III. The major contribution of this paper is to determine if Linux kernel modifications for supporting thin-streams, which MMOGs are an example of, can improve the server selection results as well as decrease the latency further for the interacting users. The Linux kernel modifications came as a result of the work of Andreas Petlund, and their integration and testing was performed by Beskow.

6.3 Nornir

Nornir is a shared-memory machine implementation of a processing framework based on Kahn Process Networks (KPNs), though no distributed implementation was ever created, these concepts also extend into the distributed domain. With Nornir our aim was to support iterative algorithms, i.e., algorithms containing cycles or feedback loops in their data-path, such as H.264/AVC, as discussed in section 2.2.3. As Nornir is based on the formalism of KPNs, it is a simpler alternative to traditional shared-memory concurrency, as it inherently supports deterministic execution, and thus composability. Synchronization of state between processes in Nornir is effectively message passing, but there are some differences in the way communication is achieved. With KPNs, all communication is achieved through infinitely sized unidirectional point-to-point channels. It is not known when the receiving process will read a message from the channel, so it is only known that it will be read at some point in time.

KPNs provide a minimal set of features to ensure deterministic behaviour, which guarantees that a program, given the same input, will behave identically on each run. This significantly eases debugging and consequently determinism ensures composability, which guarantees that components can be composed and still yield the same results.

At its core, Nornir is a shared-memory message passing based system. Explicitly defining communication can be cumbersome for irregular or complex pipelines, which are the type of pipelines we want to support with Nornir. However, unlike comparable message passing systems, as discussed in section 5.3.2, Nornir intrinsically supports deterministic execution. As such, communication patterns can be more difficult to express, but the developer can implement their code as sequential pieces of code and get the same result each time the application is executed. By removing this overhead of having to utilize low-level synchronization primitives, such as mutex', condition variables, etc., we significantly reduce the effort required by the developer. With Nornir, we consider this trade-off a viable approach.

The development of Nornir was solely performed by Zeljko Vrba with him also being the primary author of all papers. Paul Beskow acted as an auxiliary author that contributed to the discussions of ideas and definition and exploration of workloads. Also implementing an algorithm for finding cycles in graphs and identifying the pros and cons of KPNs to comparable systems.

Paper VI: Kahn process networks are a flexible alternative to MapReduce

In this article, we formalized the impact of shared-memory concurrency and identified low-level synchronization primitives as inherently difficult concepts to utilize by developers. Based on these observations, we identified prevalent methods for negating the cognitive overhead of using shared-memory concurrency. We focused on Google's MapReduce, a widely used parallel programming paradigm, in which the user is able to write sequential code using a key-value programming model, within two predefined stages, and achieve parallel execution of tasks with minimal overhead on the developer.

The rigidity of MapReduce and similar techniques, such as Microsoft's Dryad, transactional memory, etc, resulted in the identification of KPNs as a viable alternative to these approaches. KPNs support sequential coding of individual processors, composability, deterministic execution, and more importantly, arbitrary communication graphs and not a prescribed programming model, such as MapReduce.

The KPN implementation is then compared to Phoenix, a multi-core implementation of MapReduce, and to a version of MapReduce implemented in the Nornir run-time system, where the KPN approach provides considerable speed-up. Furthermore, Nornir can more naturally model a broader set of applications with minimal impact on the developer, when compared to MapReduce and Phoenix.

Paper VII: The Nornir run-time system for parallel programs using Kahn process networks

In this paper, the possibilities provided by KPNs are explored more thoroughly and rewrite of the internal mechanisms composing the run-time system implementation are performed; with the implementation of a work-stealing algorithm for process scheduling and an extensive accounting system for evaluating the performance of the run-time system.

Due to the flexibility provided by the KPN model, we identify a set of interactive multimedia applications that provide complex interaction patterns, looking at the H.264/AVC and *k*-means workloads. These workloads are essential, because they can greatly benefit from a high-level programming model that is capable of modeling arbitrary communication graphs and also require considerable computational resources. We also benchmark a random graph, to demonstrate the flexibility of the KPN programing model.

Thus, a primary contribution of this paper was to explore the flexibility, performance and scalability of applications modeled as KPNs. The primary contribution of Beskow was to the identification and evaluation of workloads and discussion of results.

Paper VIII: The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce

In this journal paper, we expand on knowledge gained in **Paper VII**. We present a modified version of the classical work-stealing algorithm and also present a novel graph partitioning algorithm for scheduling processes. As such, a primary contribution of this journal paper is the evaluation of different scheduling policies. Beskow's contribution to this paper was primarily as an auxiliary author, and in particular, some of the discussions surrounding graph partitioning.

6.4 P2G

Based on the initial ideas from Ginnungagap and Nornir, and drawbacks and shortcomings of these systems, such as Nornir's use of explicit communication channels which makes it cumbersome to use with data-parallelism, and Ginnungagap's object based model, we have designed and implemented P2G, where P2G is a high-level programming model and run-time system for distributed real-time multimedia processing. It is designed to work on continuous flows of data, such as live video streams, while still maintaining the ability to support batch workloads. P2G supports arbitrarily complex dependency graphs with cycles, branches and also deadlines to provide the required timeliness for multimedia. It also supports both data and task parallelism. P2G therefore solves a number of issues we have been aware of since first encountering them in our earlier work with Ginnungagap and the evaluation of Nornir. P2G

represents the best practice through the work on Ginnungagap and evaluation of interactive multimedia workloads as described in Nornir. The work on P2G was primarily performed by Paul Beskow and Håvard Espeland, where the focus of Beskow's contribution was on the definition of a high-level programming model and its subsequent impact on the run-time system. Espeland contributed to the high-level programming model and the efficient execution of tasks in a run-time system, where his focus was more on low-level optimizations and scheduling.

Paper IX: Distributed Real-Time Processing of Multimedia Data with the P2G Framework

In this poster at Eurosys, we presented the initial work in progress, in form of a 2-page explanation. This was then followed by a poster, which was created after the implementation and description in **Paper X** was completed. As such, we refer to the following section for a description of the content within this poster.

Paper X: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

In this SRMPD paper, we present P2G, which was born out of the observation that most distributed processing framework lack support for real-time multimedia workloads, and that either data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they can severely limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but restrict run-time processing to the domain of data parallelism [88]. Functional languages, such as Erlang [89] and Haskell [90] and the event-based SDL [91], map well to task parallelism. Programs are expressed as communicating processes either through message passing or event distribution, which makes it difficult to express data parallelism without specifying a fixed number of communication channels. With P2G we present a unified view of the knowledge gained from implementing Ginnungagap and the evaluation of Nornir and related work and successfully apply this to the formulation of a new high-level programming model and implementation of a run-time system to support the execution of workloads specified in the high-level programming model.

Paper XI: Processing of Multimedia Data using the P2G Framework

In this ACM MM demonstration paper, we implemented a motion JPEG encoder and k-means workloads using the P2G system. As such, the primary contribution of this paper was the possibility of dynamically scaling the number of threads utilized by the P2G framework, and thus the number of cores dynamically during run-time. We did this to show off the abilities of the framework and to demonstrate how the run-time system is able to scale and increase the performance of the application, in this case in form of increasing the frames per second for generated output.

6.5 Lessons learned

In the following sections we will summarize what we have learned while working on the prototypes discussed so far and what impact interactive multimedia workloads can have on the high-level programming models and run-time systems that support the development and execution of these workloads.

6.5.1 Ginnungagap

Ginnungagap was designed for migrating state, encapsulated by objects, in a distributed system by extending the object-oriented language C++ with functionality for performing remote method invocations (RMI). Efficient maintenance of references to objects in the system was then achieved by using a distributed name service. Through the development of Ginnungagap, we learned a number of valuable lessons, as summarized here.

Deferred decisions

Being able to defer critical decisions to run-time, such as scheduling and load balancing, was identified early as an important facet (*Paper I, section 1*), because interactive multimedia applications are intrinsically unpredictable; it is fundamentally impossible to predict the future load of a server or determine the interaction patterns of an application's users at compile-time or during deployment of an application. Thus, being able to dynamically change the configuration of the system during run-time is essential; by migrating tasks to optimize for data locality (*Paper III, section II*).

Decoupling programming model and run-time

Decoupling the high-level programming model, which the developer interacts with, from the run-time system is also a beneficial property (*Paper I, section 4*). This can simplify the development process by automatically generating code that integrates with the application programmer interface (API) of a run-time system by ensuring that the API is correctly used and can also make it possible to seamlessly integrate with multiple different run-time systems and their respective APIs. Furthermore, it can facilitate developers by allowing them to focus on expressing the interaction patterns between tasks and data instead of being concerned with cumbersome and error-prone task, such as synchronization, communication, scheduling and similar activities.

Distributed name service

A distributed name service (*Paper I, section 3*) supports the efficient maintenance of references to objects in a topology of processes or nodes in a distributed system. This is achieved because references to state is initially kept local and only when data is migrated, a reference update is required. While frequent migrations can lead to long chains of references, this can be alleviated by embedding the address of the calling node in the invocation and updating the name service of the requesting node upon reception of the reply. In systems where state is frequently created and destroyed, the network communication required to identify the location of state can be greatly minimized as a result.

Migration or resource locality

Being able to migrate state (*Paper III, section II*) between processes or nodes in a distributed system of resources to dynamically adapt to the changes in hardware resources and arbitrary shifts in interaction patterns of the users is fundamentally important. Optimizing the location of state or tasks relative to the interacting parties can ensure that a sufficient Quality of Experience (QoE) is achieved.

Summary

The focus of Ginnungagap was to efficiently support massively multi-player online games. However, as it is becoming common to include even more types of content in MMOGs, with some early efforts moving in the direction of augmented reality games [92], supporting a number of different content types is becoming increasingly important. While the primary focus of Ginnungagap was on MMOGs its functionality can be used by a wide range of interactive applications, from video conferencing to VoIP, where low and equal latency for the interacting parties is important.

6.5.2 Nornir

Nornir is a shared-memory machine implementation of Kahn Process Networks (KPNs). The work on Nornir was motivated by a desire to improve the debuggability of complex workloads and to provide a more flexible alternative to the existing rigid models of MapReduce and Dryad. With Nornir the aim was to support iterative algorithms, i.e., algorithms containing cycles or feedback loops in their data-path, because Nornir has the ability to model iterative algorithms it is better suited for a broader range of interactive multimedia workloads, such as real-time encoding of video streams. As Nornir is based on the formalism of KPNs, it is a simpler alternative to traditional shared-memory concurrency, and it inherently supports deterministic execution, and thus composability. With KPNs, all communication is achieved through infinitely sized unidirectional point-to-point channels with blocking read semantics.

Computationally and logically complex workloads

Due to the complex data and task inter-dependencies of modern interactive multimedia workloads, as exemplified by the H.264/AVC workloads defined in section 2.2.3, finding a suitable abstraction for specifying the interaction between communication and computation is necessary. Furthermore, a number of interactive multimedia workloads are computationally intensive, and the efficient execution of these workloads is therefore necessary.

Compositionality

Compositionality (*Paper VI, section 1*) means that a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. Being able to create interactive multimedia applications through a composition of components can greatly help reduce the complexity of managing these workloads, because the independent components can be checked for correctness before they are assembled into a full system.

Determinism

With deterministic execution (*Paper VI, section 4*), a task is guaranteed to always produce the same output given the same input. A non-deterministic execution of tasks can not guarantee

this, which can lead to observable differences in the output computed by a task. This is highly beneficial for concurrent execution, because it makes it easy to reason about the correctness of the application and greatly facilitates the debugging process.

Explicit dependency graph

Explicit dependency graphs (*Paper VII, section II*) provide an intuitive representation of the interaction between tasks and data in a workload, where the vertices represent the computation or tasks, and the edges represent the communication channels between the tasks. With explicit dependency graphs, the developer is responsible for manually specifying the communication channels between tasks. This abstraction makes it possible to reason about the structure of the application, and it also makes it easier to add additional functionality as the state is explicitly communicated between tasks meaning that a task has to add additional communication channels to new tasks.

Summary

Nornir is based on KPNs, and their formalism, which is based on some unreasonable assumptions, such as unlimited channel capacities, which is not the case in modern computer systems, due to limits on channel capacities, which are required because of limited memory. Artificial dead-locks can then occur in the computation. When extending Nornir to a distributed system, a distributed dead-lock detection mechanism would have to be implemented. Depending on the size of the Kahn network, this could cause inordinate amounts of delays.

6.5.3 P2G

Based on the initial ideas from Ginnungagap and Nornir, and drawbacks and shortcomings of these systems, such as Nornir's use of explicit communication channels which makes it cumbersome to use with data-parallelism, and Ginnungagap's object based model, we have designed and implemented P2G, where P2G is a high-level programming model and run-time system for distributed real-time multimedia processing. The idea of P2G was born out of the observation that most distributed processing frameworks lack support for real-time multimedia workloads, and that data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they can severely limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but restrict runtime processing to the domain of data parallelism [88]. Functional languages such as Erlang [89] and Haskell [90] and the event-based SDL [91], map well to task parallelism. Programs are expressed as communicating processes either through message passing or event distribution, which makes it difficult to express data parallelism without specifying a fixed number of communication channels.

Named state

In the context of a running application, state is a term used to describe all the information stored at a given point in time. Named state (*Paper X, section III*) then refers to shared and accessible state, while unnamed state implies that there is no way of directly accessing the

current state of the application. Interactive multimedia algorithms require named state, because this provides an intuitive way of specifying the interaction between tasks and data. The reason for this is because these algorithms are typically defined in terms of transformations over data. As such, reading and writing from a shared data structure, reachable from all tasks, makes it easier to define these interactions.

Write-once semantics

Write-once semantics (*Paper X, section III*) support deterministic execution of tasks interacting through named state. This is achieved because a variable is guaranteed to only be written to once and can therefore be read safely by any depending tasks without having to consider race-conditions, i.e., that multiple concurrent tasks access and modify shared state resulting in observable differences in the computed output. This is beneficial, because there is then no requirement of explicitly defining the interactions between tasks and data.

Implicit dependency graph

Explicit dependency graphs require the developer to manually specify the communication channels between tasks. From a developers perspective, formulating these explicit communication channels is cumbersome, time consuming and error-prone in addition to not providing natural support for interactive multimedia algorithms. Instead, implicitly defining the relationships between tasks and data through fetch and store statements to global data structures provides a more natural thought pattern (*Paper X, section V*). This then gave us the possibility of harnessing the power of explicit communication through a high-level abstraction that provides implicit semantics.

6.5.4 Overall

With Ginnungagap we looked at the possibilities provided by applications that could be inherently decomposed into components that can run in parallel can be accomplished by a developer and supported by efficient execution on a run-time system.

In our multimedia scenario, Nornir improved on many of the shortcomings of the traditional batch processing frameworks, like MapReduce and Dryad. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs support also arbitrary communication graphs (with cycles or iterations), while frameworks like MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs).

However, Nornir is task-parallel, and data-parallelism must be explicitly added by the programmer. Furthermore, as a distributed, multi-machine processing framework, Nornir still has some challenges. For example, the message-passing communication channels, having exactly one sender and one receiver, are modeled as infinite FIFO queues. In real-life distributed implementations the queue length is limited by available memory. A distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually. This requires the developer to think differently than for other distributed frameworks.

A major source of non-determinism in other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. The source of this non-deterministic

behavior can be removed by adopting strict write-once semantics for writing to memory [93]. Languages that take advantage of the concept of single assignment include Erlang [89] and Haskell [90]. It enables schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once-semantics are well-suited for a scheduler's dependency analysis, it is not straight-forward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multi-dimensional arrays of data (e.g., pixels in a picture) and provide frequently very intuitive data partitioning opportunities (e.g., 8x8-pixel macro-blocks of a picture). Prominent examples are the computation-heavy MPEG-4 AVC encoding [94] and SIFT [95] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without loosing write-once-semantics.

Flexible partitioning requires the processing of clearly distinct data units without side-effects. The idea adopted for P2G is to use *kernels* as in stream processing [7,96]. Such a kernel is written once and describes the transformation of multi-dimensional fields of data. Where such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which can analyze dependencies and guarantee fully deterministic output independent of order due to the write-once semantics of fields.

Together, these observations determined four basic ideas for the design of P2G:

- The use of *multi-dimensional fields* as the central concept for storing data in P2G to achieve straight-forward implementations of complex multimedia algorithms.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *write-once semantics* to such fields to achieve deterministic behavior.
- The use of *runtime dependency analysis* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

Within the boundaries of these basic ideas, P2G should be easily accessible for programmers who only need to write isolated, sequential pieces of code embedded in kernel definitions. The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to fetch slices of a field in as fine a granularity as possible, supporting data parallelism.

With P2G we present a two-layered model for approaching the task of synchronizing state in a distributed system. Our observations from implementing frameworks for interactive multimedia applications have shown that from a developer's perspective, a shared-memory model is the most efficient and least invasive. It allows for the developer to think of communication in familiar terms, by simply indexing a multi-dimensional array. As such, the communication is handled implicitly in P2G.

A negative side-effect of implicit communication is typically that additional messages are added to the communication, to ensure consistency across the nodes. With P2G, however, we translate the implicit communication to explicit communication, effectively removing this overhead. As such, P2G has knowledge of the dependencies of a given field or slices within it.

So, when data is written to a position this will automatically be forwarded to its dependents. A high-level scheduler is responsible for creating these subscriptions.

In similar MP systems, the computation, once it has been dispatched, remains static. With P2G, we have designed for migration and replication of data inherently. As such, the high-level scheduler can move data and tasks within the system, and update the communication transparently. Since the fields in P2G are write-once, we can freely replicate data throughout the computation, without having to consider consistency problems.

Similarly, because of the semantics of the fields and kernels, we can ensure deterministic execution of workloads. As stated by Lee in [97],

As such, by dividing the abstract representation presented to the developer, and the transformations performed by the underlying run-time system, we are capable of supplying efficient synchronization of state in a distributed system.

7 | Conclusion

Interactive multimedia applications are an emerging domain of applications, consisting of computationally intensive workloads with complex task and data interdependencies. Programming model and run-time system support can greatly aid in the efficient implementation and execution of these applications, making these matters a primary concern. This is important, because parallel, heterogeneous and distributed resources provides the required resources to scale in terms of performance requirements, but are magnitudes more difficult to develop for then their sequential counter-parts. Providing a high-level programming model that creates abstractions that are intuitive to the developer are then vital, while a run-time system, per architecture, can then ensure that the available resources are efficiently utilized.

In this thesis we have addressed these issues through extensive prototyping through the implementation of Ginnungagap [15–19], evaluation of Nornir [20–22] and culminating in the implementation of P2G [23–25], whose ideas and concepts for formulating a high-level programming model and subsequent run-time system contribute to the development process of interactive multimedia applications.

7.1 Fulfillment of research objectives

The research objectives were initially defined in the introduction of this thesis (section 1.5) and are reiterated within their respective subsections for completeness. We will now evaluate to what extent we have successfully achieved these research objectives.

7.1.1 RO1: Workloads

RO1 *Workloads:* Determine what parallelization opportunities exist in interactive multimedia application workloads. Extract the unique characteristics of these workloads and determine their potential impact on subsequent investigations.

The research in this thesis has been driven by the incremental refinement of ideas through extensive prototyping. This has also been the case for the workload evaluation, where additional interactive multimedia workloads have been added throughout the work in this thesis and has subsequently had an impact on the parallelization opportunities identified within these workloads, as summarized in chapter 2, and to the expansion of properties unique to these applications.

With Ginnungagap we investigated massively multi-player online games (MMOGs), which are primarily characterized by their large numbers of concurrently interacting users, strict latency requirements and irregular usage patterns (*Paper II, section 2*). Within these applications

we determined that application level partitioning (section 2.1.1), provides a parallelization opportunity. This comes as a result of MMOGs being naturally decomposable systems (*Paper II, section 2.3*), where the interacting parties and their related state can be decomposed as a result of the application itself. This property can therefore be extended to other applications that exhibit this property.

With Nornir and P2G, we broadened the scope of workloads to include more complex workloads, focusing on the H.264/AVC (section 2.2.3) (*Paper VII, section IV*), and k-means (section 2.2.2) (*Paper VI, section 6.3*) workloads. These workloads distinguish themselves because of intricate interactions between tasks and data, commonly forming complex and arbitrary processing graphs (section 4.1.2); containing cycles and feedback loops (section 3.3.4). We identified that these workloads commonly support a variety of parallelization opportunities, ranging from data and task parallelism to nested and pipeline parallelism or combinations of all of these (section 3.3.3). Being able to fully express all these forms of parallelism is therefore essential to fully exploit their inherit parallelism.

7.1.2 RO2: Abstractions

RO2 *Abstractions:* Evaluate existing parallel programming models and extract concepts that are relevant to the criteria defined in RO1 and formulate a composition of concepts on this basis.

The evaluation of workloads provides a set of requirements that impact the formulation of a high-level programming model, which is used for expressing the workload functionality and its inherit parallelism. The creation of such a new programming model is necessarily founded on an evaluation of existing programming model concepts and best practices; identified through years of research and practical application. Furthermore, the formulation of a high-level programming model should adhere to qualities one deem important (section 3.2). On this basis we identified a number of important programming model concepts, as summarized in chapter 3.

Due to the complex data and task inter-dependencies of modern interactive multi-media workloads, as exemplified by the H.264/AVC workloads (section 2.2.3), it is necessary to find a suitable abstraction for specifying the interaction between communication and computation. Explicit dependency graphs, as evaluated in Nornir, then provide an intuitive representation of the interaction between tasks and data in a workload, where the vertices represent the computation or tasks and the edges represent the communication channels between the tasks (*Paper VI, section 3*). With Nornir, we gained an understanding of the usefulness of this representation through the formulation of explicit dependency graphs, i.e., the developer is responsible for manually specifying the communication channels between tasks. However, efficiently formulating these explicit communication channels and their subsequent graph representation is cumbersome, time consuming and error-prone.

Another primary contribution of Nornir was the identification of determinism (*Paper VI, section 3*)(section 3.3.2) as a desirable quality. A deterministic computation will always produce the same output given the same input, which can greatly facilitate the developer during the debugging process. Determinism also supports compositionality (section 3.3.5). In this respect, compositionality guarantees that the behavior of the whole is a logical combination of the behavior of the individual parts (*Paper VI, section 3*).

A major contribution of Ginnungagap was the identification of the importance of decoupling (*Paper I, section 4*)(section 4.1.1), where decoupling provides the ability to separate two

components from each other, such as decoupling the high-level programming model from the run-time system. Decoupling can therefore affect the interaction between components in a number of ways, such as the way the high-level programming model is exposed to the developer, as discussed in section 3.4. In Nornir, we did not use this approach, which made the task of specifying the explicit communication channels between tasks cumbersome and error-prone.

With P2G, by decoupling the high-level programming model from the run-time system, we were able to provide abstractions that could greatly facilitate the developer. Nornir used unnamed state to synchronize the interaction between tasks and data, by explicitly defining the communication channels. In P2G, however, we explored the concept of named state (section 3.3.1), i.e., a global data structure accessible to all tasks. This made it possibly to implicitly extract the processing graph of the workload (*Paper X, section V*)(section5.3.1). For multimedia algorithms, named state provides for more natural interaction with the data and makes it possible to define the interaction patterns implicitly. A primary contribution of P2G was then to ensure determinism and compositionality while using named state (*Paper X, section III*), which was achieved by using write-once semantics.

7.1.3 RO3: Execution

RO3 *Execution:* Explore the feasibility of supporting the execution of interactive multimedia workloads given the definition of a workload in a high-level parallel programming model as formulated by RO2.

Given a definition of an interactive multimedia workload in a high-level programming model, executing this workload given a topology of hardware resources becomes a primary objective.

With Ginnungagap, we first identified the importance of deferring important decisions to run-time (*Paper II, section 3*), where the primary purpose in Ginnungagap was to achieve load-balancing and reduce latencies by optimizing the resource locality. As a consequence, we explored the concepts of migration (*Paper II, section 4*), to intrinsically support the movement of state in a system of distributed objects. We also identified the need for an efficient way to maintain references; using a distributed name service (*Paper II, section 4.1*)(section 5.2). The concepts of migration and a distributed name service were never applied to P2G or Nornir in the context of this thesis, because it was beyond the scope of the thesis, but P2G, in particular, was designed in such a way that these techniques could be applied.

With P2G and Nornir, we revisited the importance of deferring decisions to run-time by looking at the importance of being able to defer scheduling decisions to run-time, as discussed in section 5.3.2. A major contribution in this context is the ability to dynamically adapt to the existence of multiple workloads running on the same hardware or to the addition or removal of hardware.

Given a high-level specification of a workload, a primary objective is to translate the workload into tasks that are consumable by the run-time system. In Ginnungagap, objects encapsulated state and computation and events were based on remote method invocations (section 5.3.2), i.e., these objects could be relocated based on algorithms implemented in the run-time system to ensure resource locality. In Ginnungagap, there was no way of extracting the processing graph of the application, either explicitly or implicitly, primarily because this was not necessary due to the relatively simple interaction patterns. Furthermore, partitioning the application was a primary method for ensuring scalability.

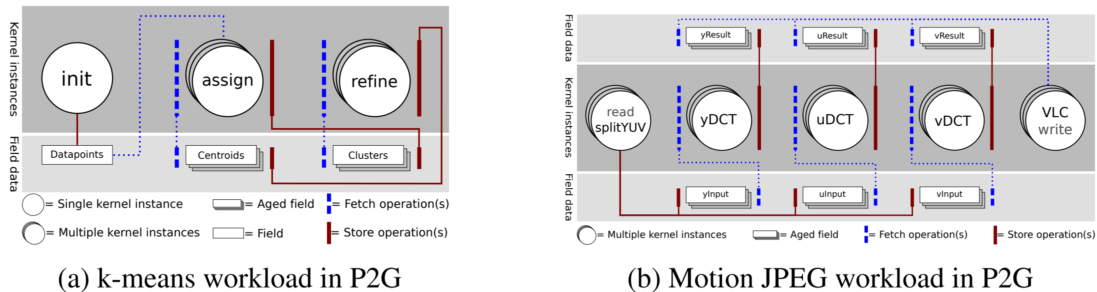
With Nornir and P2G, the focus moved towards supporting more complex interaction patterns due to more generic workloads, such as H.264. The processing graph abstraction therefore provided an intuitive way of defining the interaction between tasks and data. A major contribution of both Nornir and P2G was through the translation of these high-level definitions of tasks to a light-weight task model (section 5.3.1). The primary objective with these high-level models was to expose as much of the inherent parallelism of the workload as possible and then have each parallel unit represented as a light-weight task (*Paper X, section V*). Light-weight tasks provide a smaller granularity of a schedulable unit, and they also make it a lot easier to overlap communication and computation (section 5.1), which is important to achieve scalability (*Paper X, section VIII*).

In both Nornir and P2G, theses light-weight tasks were never scheduled until all data for that task was available. The queue of tasks waiting to be executed would therefore be populated with only ready tasks, as discussed in section 5.3.2.

7.1.4 RO4: Validation

RO4 *Validation:* Validate the concepts extracted from RO2 and RO3 and their applicability to the requirements of RO1 through prototyping.

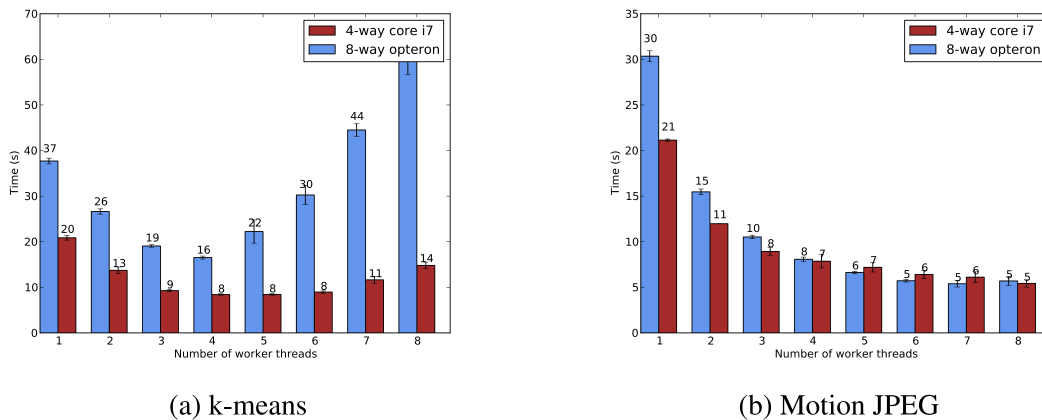
The prototype implementation of Ginnungagap and the evaluation of properties in Nornir and related work have provided valuable input into the extraction and validation of key programming model concepts, as discussed in chapter 3. We have successfully applied these concepts to the formulation of a new high-level programming model and run-time system as implemented with P2G (*Paper X*). In figure 7.1, we see a high-level view of the multimedia workloads *k*-means and Motion JPEG, as implemented in P2G. This demonstrates the use of kernel definitions, as exemplified by *assign, refine, yDCT*, etc., and the possibility of having multiple kernel instances operating in parallel over named state, as exemplified by *Centroids, Clusters*, etc.



(a) k-means workload in P2G (b) Motion JPEG workload in P2G

Figure 7.1: Validation of P2G

Furthermore, in our evaluation of these workloads and their execution in P2G, we demonstrate the scalability of this solution with regards to local utilization of resources, as seen in figure 7.2.



(a) k-means (b) Motion JPEG

Figure 7.2: Validation of P2G

7.2 Critical assessments

One of the objectives of this thesis was to formulate a high-level programming model for implementing interactive multimedia algorithms. A correlating goal was to implement a run-time system capable of efficiently executing a variety of hardware architectures, including multi-core CPUs, GPUs and heterogeneous and distributed hardware resources.

In this thesis, we have contributed to the implementation of a multi-core, x86 run-time system capable of executing these workloads, which demonstrates the feasibility of the solution. Though the exploration of novel scheduling mechanisms proved to be beyond the scope of this thesis, a number of ideas were centered around the idea of dynamically altering the granularity of task and data parallelism, to reduce the overhead of scheduling small tasks individually. Another goal was to exploit the resources made available from heterogeneous and distributed systems, implementing efficient run-time systems for each architecture. An initial idea was to use graph partitioning mechanisms, using a cost-function based on reducing communication, latencies, etc., to achieve an efficient scheduling in distributed systems. Within the context of the project the field of scheduling is being examined by another thesis.

An additional idea was to exploit just-in-time compilation to increase the performance of individual tasks by dynamically merging tasks or by generating code that could integrate with hardware resources. One could then distribute byte-code to nodes in a distributed system and have each node compile the code to be runnable on the available hardware resources, for example a multi-core machine with a GPU.

The high-level programming model is implemented using language constructs defined by the authors, none of which have a sounds basis in language design or compiler techniques.

We have made the assumption within this thesis that all code is shared between the interacting computing nodes. We are aware of the problems related to distributing code in large clusters of compute nodes, but have not considered this a fundamental part of our system. This decision was based on the fact that we operate with preconfigured systems that process streams of data over extended periods of time. Thus, making the benefit of considering such solutions negligible to our current scenarios.

7.3 Future work

There exists a number of possible directions for continuing the work started with this thesis. From a scheduling perspective there exists a wide variety of opportunities, where one possibility is to start with the formulation of new low-level schedulers. A low-level scheduler in this scenario is responsible for ensuring the local utilization of resources is optimal, i.e., that the resources within a single node are fully utilized.

Implementing a distributed version of the run-time system would also be highly beneficial, which also includes the necessity for formulating a high-level scheduler, i.e., a scheduler that is responsible for partitioning and distributing a workload across the nodes in the topology.

Run-time systems for utilizing heterogeneous resources is also a primary objective. One could then utilize graphical processing units, Cell BE, etc, or non-cache coherent architectures like Intel's SCC and Xeon Phi.

One could also look at the possibilities of creating fat binaries, so a workload could be packaged into one unit and then utilized by a number of different run-time systems.

Bibliography

[1] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[2] S. Hiranandani, K. Kennedy, and C.W. Tseng. Evaluation of compiler optimizations for fortran d on mimd distributed memory machines. In *Proceedings of the 6th international conference on Supercomputing*, pages 1–14. ACM, 1992.

[3] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, 1995.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[5] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. Sisal: streams and iteration in a single-assignment language. language reference manual, version 1. 1. Technical report, Lawrence Livermore National Lab., CA (USA), 1983.

[6] M.M.T. Chakravarty, R. Leshchinskiy, S.P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[7] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. MIT press, 1994.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*, volume 1. MIT press, 1999.

[10] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[11] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. 1988 Int. Conf. Parallel Processing*, volume 2, pages 94–101, 1988.

[12] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. *ACM SIGOPS Operating Systems Review*, 23(5):211–223, 1989.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[15] P.B. Beskow, K.H. Vik, P. Halvorsen, and C. Griwodz. Latency reduction by dynamic core selection and partial migration of game state. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 79–84. ACM, 2008.

[16] P.B. Beskow, G.A. Erikstad, P. Halvorsen, and C. Griwodz. Evaluating ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, page 10. IEEE Press, 2009.

[17] P.B. Beskow, K.H. Vik, P. Halvorsen, and C. Griwodz. The partial migration of game state and dynamic server selection to reduce latency. *Multimedia Tools and Applications*, 45(1):83–107, 2009.

[18] Paul Beskow, Pål Halvorsen, and Carsten Griwodz. Latency reduction in massively multiplayer online games by partial migration of game state. In Denise Oram Vic Grout and Rich Picking, editors, *Second International Conference on Internet Technologies and Applications*, pages 153–163, University of Wales, NEWI, Wrexham, UK., September 2007. Centre for Applied Internet Research (CAIR).

[19] Paul Beskow, Andreas Petlund, Geir Erikstad, Carsten Griwodz, and Pål Halvorsen. Reducing game latency by migration, core-selection and tcp modifications. *Inderscience International Journal of Advanced Media and Communication (IJAMC)*, 4(4):343–363, 2010. Special issue on selected papers from NetGames.

[20] Zeljko Vrba, Paul Beskow, Pål Halvorsen, and Carsten Griwodz. Kahn process networks are a flexible alternative to mapreduce. In NA, editor, *Proceedings of 11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 154–162. IEEE Computer Society, 2009.

[21] Ž. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen. The nornir run-time system for parallel programs using kahn process networks on multi-core machines - a flexible alternative to mapreduce. *The Journal of Supercomputing*, pages 1–27, 2010.

[22] Z. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, and D. Johansen. The nornir run-time system for parallel programs using kahn process networks. In *Network and Parallel Computing, 2009. NPC'09. Sixth IFIP International Conference on*, pages 1–8. IEEE, 2009.

[23] Paul Beskow, Håvard Espeland, Håkon Kvale Stensland, Preben N Olsen, Ståle B Kristoffersen, Espen Angell Kristiansen, Carsten Griwodz, and Pål Halvorsen. Distributed real-time processing of multimedia data with the p2g framework. EuroSys 2011, April 2011.

[24] Håvard Espeland, Paul Beskow, Håkon Kvale Stensland, Preben N Olsen, Ståle B Kristoffersen, Carsten Griwodz, and Pål Halvorsen. P2g: A framework for distributed real-time processing of multimedia data. In Jang-Ping Sheu and Cho-Li Wang, editors, *Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2011 International Conference on Parallel Processing Workshops*, pages 416–426. IEEE, September 2011.

[25] Paul Beskow, Håkon Kvale Stensland, Håvard Espeland, Espen Angell Kristiansen, Preben N Olsen, Ståle B Kristoffersen, Carsten Griwodz, and Pål Halvorsen. Processing of multimedia data using the p2g framework. In Nicu Sebe Hari Sundaram, Wu-Chi Feng, editor, *Proceedings of the 19th ACM international conference on Multimedia*, pages 819–820. ACM, 2011.

[26] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.

[27] J.A. Hartigan and M.A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[28] I. Draft. Recommendation and final draft international standard of joint video specification (itu-t rec. h. 264| iso/iec 14496-10 avc). *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVT- G*, 2003.

[29] Iain Richardson. White paper: A technical introduction to h.264/avc. *Vcodex Ltd.*, 2011.

[30] D.A. Kerr. Chrominance subsampling in digital images. *The Pumpkin,(1), November*, 2005.

[31] L. Aimar, L. Merritt, E. Petit, M. Chen, J. Clay, M. Rullgrd, C. Heine, and A. Izvorski. x264-a free h264/avc encoder. *Online (last accessed on: 21/02/12): http://www. videolan. org/developers/x264. html*, 2005.

[32] D. Szafron, J. Schaeffer, and University of Alberta. Dept. of Computing Science. An experiment to measure the usability of parallel programming systems. *Concurrency Practice and Experience*, 8(2):147–166, 1996.

[33] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 35–35. IEEE, 2005.

[34] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley, 1995.

[35] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

[36] P. Van Roy. Programming paradigms for dummies: what every programmer should know. *New Computational Paradigms for Computer Music*, 2009.

[37] D.P. Delorey, C.D. Knutson, and C. Giraud-Carrier. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*, 2007.

[38] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[39] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[40] H.W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G.J. Michaelson, R. Peña, S. Priebe, et al. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[41] R.L. Bocchino Jr, V.S. Adve, S.V. Adve, and M. Snir. Parallel programming must be deterministic by default. *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4, 2009.

[42] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. *Conference on Programming Language Design and Implementation: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, 21(24):258–266, 1996.

[43] R.L. Bocchino Jr, V.S. Adve, D. Dig, S.V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. *ACM SIGPLAN Notices*, 44(10):97–116, 2009.

[44] G. Kahn. *The semantics of a simple language for parallel programming*. North-Holland, 1974.

[45] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[46] Y. Bu, B. Howe, M. Balazinska, and M.D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[47] J. Armstrong, R. Virding, C. Wikstr, M. Williams, et al. *Concurrent programming in ERLANG*. Prentice Hall, 1996.

[48] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[49] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[50] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[51] I. Foster and C. Kesselman. Language constructs and runtime systems for compositional parallel programming. *Parallel Processing: CONPAR 94-VAPP VI*, pages 5–16, 1994.

[52] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M.M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[53] M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal-nested data parallelism in haskell. *Euro-Par 2001 Parallel Processing*, pages 524–534, 2001.

[54] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, pages 207–216, 1995.

[55] Charles Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51:244–257, 2010. 10.1007/s11227-010-0405-3.

[56] M.E.J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[57] J.P. Shen and M.H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. McGraw-Hill Higher Education, 2005.

[58] S.L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175, 1989.

[59] L.M. Silva and R. Buyya. Parallel programming models and paradigms. *High Performance Cluster Computing: Architectures and Systems*, 2:4–27, 1999.

[60] R. Griesemer, R. Pike, K. Thompson, et al. The go programming language, 2009.

[61] D.G. Murray and S. Hand. Scripting the cloud with skywriting. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 12–12. USENIX Association, 2010.

[62] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proceedings of the 1999 ACM symposium on Applied computing*, pages 156–165. ACM, 1999.

[63] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41:37–49, April 2007.

[64] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[65] T. White. *Hadoop: The definitive guide*. Yahoo Press, 2010.

[66] R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on dynamic languages*, page 1. ACM, 2008.

[67] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14. USENIX Association, 2008.

[68] X.J. Yang, X.K. Liao, K. Lu, Q.F. Hu, J.Q. Song, and J.S. Su. The tianhe-1a supercomputer: its hardware and software. *Journal of Computer Science and Technology*, 26(3):344–351, 2011.

[69] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[70] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[71] K.M. Chandy and C. Kesselman. Cc++: a declarative concurrent object-oriented programming notation. *Research directions in concurrent object-oriented programming*, pages 281–313, 1993.

[72] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[73] M.M. Rafique, B. Rose, A.R. Butt, and D.S. Nikolopoulos. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[74] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.

[75] R.M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE, 2009.

[76] G. Agha. An overview of actor languages. *OOPWORK: Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming: Yorktown Heights, New York, United States*, 9(13):58–67, 1986.

[77] A. Silberschatz, P.B. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley, 1998.

[78] A.S. Tanenbaum and A.S. Woodhull. *Operating systems: design and implementation*, volume 68. Prentice Hall, 1997.

[79] J.S. Snyder, D.B. Whalley, and T.P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, 1995.

[80] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.

[81] M. Zaharia. Job scheduling with the fair and capacity schedulers. *Hadoop Summit*, 2009.

[82] M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[83] T.B. Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.

[84] R. Orfali and D. Harkey. *Client/server programming with Java and CORBA*. John Wiley & Sons, Inc., 1998.

[85] A.S. Grimshaw. Easy-to-use object-oriented parallel processing with mentat. *Computer*, 26(5):39–51, 1993.

[86] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[87] J.B. Carter et al. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, 1995.

[88] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[89] Joe Armstrong. A history of Erlang. In *Proc. of ACM HOTL III*, pages 6:1–6:26, 2007.

[90] Paul Hudakand John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proc. of ACM HOTL III*, pages 12:1–12:55, 2007.

[91] ITU. *Z.100*, 2007. Specification and Description Language (SDL).

[92] Ohan Oda and Steven Feiner. Rolling and shooting: two augmented reality games. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '10, pages 3041–3044, New York, NY, USA, 2010. ACM.

[93] R.S.N. Arvind, R.S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, 1989.

[94] ISO/IEC. *ISO/IEC 14496-10:2003*, 2003. Information technology - Coding of audio-visual objects - Part 10: Advanced Video Coding.

[95] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[96] Nvidia. Nvidia cuda programming guide 3.2, August 2010.

[97] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

Part II

Research Papers

Paper I

Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State

Paul B. Beskow, Pål Halvorsenand Carsten Griwodz

Published: Second International Conference on Internet Technologies and Applications (ITA), ed. by Vic Grout, Denise Oram and Rich Picking, pp. 153–163, University of Wales, NEWI, Wrexham, UK., Center for Applied Internet Research (CAIR), 2007

Author contribution: Beskow was the driving force behind this article. The architecture was developed solely by Beskow, who was also the primary investigator.

Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State

Paul B. Beskow[1], Pål Halvorsen[1,2], Carsten Griwodz[1,2]
[1]IFI, University of Oslo, Norway [2]Simula Research Laboratory, Norway
Email: {paulbb, paalh, griff}@ifi.uio.no

Abstract

With the increasing popularity of massively multi-player online games (MMOGs), developers are continually forced to deal with the conflicting requirements of supporting a large number of concurrent users, while simultaneously providing low latency. As a result, a common way of distributing load is by dividing the virtual environment into logical regions. Geographically coupled users in such regions can be distinguished by analyzing IP addresses, RTTs or similar. This paper proposes the architecture for a decentralized middleware capable of utilizing such information, with the intent of decreasing the overall latency for the majority of users in that region. The latency reduction is accomplished by migrating a game region to a server closer in locality to the users, thereby lowering the response time of remote procedure calls. Due to the characteristics of MMOGs, the middleware implements a distributed name service, made possible by activating the system from a single node in the system.

1 Introduction

In recent years, massively multi-player online games (MMOGs) have become increasingly popular among consumers. According to the Entertainment Software Association [1], the number of gamers who play online games has increased from 31 percent in 2002 to 44 percent in 2006. In correspondence, the number of subscribers to MMOGs has steadily increased since 1997, exceeding 13 million in 2006 [2]. These applications enable a user to form and maintain social bonds, using the virtual environment as an interface. The persistent, continuous and interactive nature of this genre of games has largely contributed to this success. Incidentally, the success due to interactivity also raises the most challenging system requirement, i.e, *low latency* for all users.

MMOGs allow thousands of users to concurrently interact in a persistent virtual environment. A significant characteristic of this type of application is its lack of resilience towards network transmission delays [3–5]. For the users to have a consistent view of the world, the events of the game need to be distributed as fast as possible. With a large number of users, the quantity of events to distribute are considerable. This causes the capacity of the servers to deteriorate rapidly, which in turn lowers the quality of service. To support these virtual environments, with such considerable numbers of interacting entities, there is a need for an efficient architecture able to handle the load generated. As a result, MMOGs commonly deploy an exclusive instance of the virtual environment on a single, centralized, distributed system, such as a cluster, grid or mainframe. One such instance of the game world is commonly referred to as a shard, a concept inherited from Ultima Online [6].

There are varying strategies for how these game worlds are deployed. EVE Online [7], for instance, only utilizes a single shard, to which all subscribers connect, and this single cluster can handle 30,000 concurrent users. In contrast, games such as World of Warcraft (WoW) [8] use multiple shards, which can support approximately 3,000 concurrent users per shard. As the load increases, more shards are made available. Alternately, shards are merged if the amount of users becomes significantly lower. A main difference is that the users of EVE Online will have the ability to interact with all users in the same environment, while in WoW, the users will have to agree on a server to connect to in order to play together.

Irrespective of the number of users supported per shard, it is common to divide the game world into regions (also referred to as cells). This is to ensure an even distribution of the load and is accomplished by reducing the number of entities that updates need to be issued to. Most commonly, these regions are divided statically. Players then connect to the server managing the region of the game world where their character is situated. As the player moves between regions, the player's data is migrated to the server handling that region. However, as far as the user is concerned, the game world is a single entity.

The users interacting in MMOGs commonly are from widely different areas of the world. In many distributed systems, the effect of this is not necessarily prevalent, mainly because the architecture of the application can adapt to this by distributing users to servers accordingly. With MMOGs, users cannot necessarily be separated to accommodate this, because of the interaction that occurs with other users in the virtual environment. As a result, users cannot be placed in the virtual world according to their physical location. It becomes apparent that the static region based architecture on a centralized cluster, while efficient and relatively easy to maintain, does not cater to the varying geographical locations of the users. It can be argued that games, such as WoW, to a certain degree take this into account. They have shards located at multiple locations across the world, where users generally connect to one close in proximity. The distribution of users, however, occurs as a result of the availability of servers, not because of how the middleware is implemented. An ideal example of this is EVE, with its single shard structure, where all users are connected to the same centralized cluster. Users from all over the world interact in the same logical regions, with one region hosted by a single server. The result is an architecture which cannot adjust itself to the difference in latency among its users.

Thus, it would be better to have a virtual world where the regions could be managed by nodes geographically located closer to the majority of the users. In this context, a recent study [9] of the MMOG Anarchy Online [10], analyzing the RTTs in traces from one of several hundred regions composing the virtual environment, three distinct groupings of users were revealed. Based on the location of the server, these were USA, Asia and Europe. It is safe to assume that one of these groups will be dominant, depending on the time of day. Thus, the assumption is that by analyzing the latency of users in a region from the virtual environment, one can determine where they are approximately located geographically. Similarly, one could use the IP addresses of the users, if available, to obtain this information. Though with a lot of research being done on integrating peer to peer based systems into MMOGs [11], there is no guarantee that such information is readily available. As far as we can determine, there has not been done a lot of research into the possibilities connected to load-balancing the regions of an MMOG based on the geographical location of users currently located within it. Most of the research is focused on effective load-balancing within a centralized cluster, by dynamically re-locating regions based on overall load. Therefore, this paper proposes the architecture of a middleware which will allow for the development of MMOGs which are aware of the physical locality of its users. The intent is to lower the response time of remote procedure calls for the majority of users connected to a given region, which in turn should lower the overall latency. This is accomplished by migrating the region to a server closer in physical locality.

Our middleware is based on a distributed model, where a single node acts as the point of initialization. That is to say, a single server in the system initializes all communication and object creation, which is necessary in order for objects in our system to be located after migration. This is a result of the way we implement our name service (see section 3). As it becomes necessary to migrate objects from one server to another, objects which share common characteristics, such as belonging to an application-defined region, are added to migration groups. These migration groups are formed on the basis of migration policies, which will select objects, managed by that node, based on a set of criteria. After the selection process, the objects in these groups are migrated to the server selected by the application. One possibility is to migrate based on geographical locations and the respective

latencies. A goal of this project is to implement this middleware in order to see if placing objects in such a manner will benefit the majority of users in terms of latency. The architecture is designed in such a way that load balancing can be performed based on any number of requirements. If the load on the server becomes too large, another migration policy can be activated. As such, migration can occur implicit or explicit. In this scenario, an explicit migration is performed when a user moves from one region to another. An implicit migration can be of two types, reactive or preemptive. A reactive policy is issued in response to scenarios such as high load on the server, which left on its own can cause the game to become unplayable. In contrast, a preemptive policy is used in an attempt to improve performance, e.g., by moving users in a region to a server closer in locality.

2 Related Work

To handle the large number of concurrently interacting entities in a virtual environment, it is common practice to use a static, region based partitioning scheme. The virtual world is thus divided into smaller, more manageable parts, where each region is hosted on a single server in the cluster. Some implementations allow several regions to be hosted on a server, such as Anarchy Online [10], while others are more conservative and allow only for one region per server, such as Second Life [12]. A widely accepted problem with the static partitioning scheme is that it does not take into account the dynamic nature of MMOGs. Even if the static partitioning is based on population density trends, and arranged to accommodate this, it is still susceptible to imbalances due to unforeseen events. Thus, a lot of research has been done on how to improve the flexibility of these partitioning schemes, and consequently, algorithms for efficiently distributing entities and regions. This research, however, does not address how the locality of the users, in relation to a server, will affect latency.

Turck et al [13] have investigated the effects of dividing a game world into dynamic micro-cells. A study with a similar background is performed by Duong et al [14]. Such micro-cells can be reassigned to servers in a cluster if the load on the server they are currently residing on becomes too large. Three different load-balancing algorithms were used, none of which factored in locality of users, and the number of micro-cells supported per server varied. The test was done on a centralized cluster. The conclusion was that a dynamic approach is preferable, because it will decrease the chance for bottlenecks and lower the overall latency.

Another approach to solving the problems with static partitioning is through maintaining consistency by limiting updates based on an area of interest. IBM has developed a middleware for distributed games called Matrix [15] using this approach. It is based on the observation that MMOGs are nearly decomposable systems, and as such, it is usually sufficient to update players with only those events that occur in their zone of visibility. Matrix thus provides pockets of locally-consistent state. Results show that Matrix outperforms static partitioning schemes when the workload exhibits unpredictable and dynamic skews. Matrix makes use of region based partitioning as an underlying foundation, but this is for the purpose of easily distributing the virtual world across multiple servers. Matrix is also intended for a centralized cluster of servers. Another middleware which implements this area of interest type partitioning is the Colyseus system [16], but this system is designed for first person shooter games and does not utilize the concept of regions.

In summary, the work on static and dynamic partitioning consider server load in a centralized cluster and not latency due to the geographical location of users. Most of the research tries to optimize the partitioning of the virtual environment into regions which can dynamically accommodate hot-spots. These regions can be user centric, in the area of interest approach, or area centric, in the micro-cell approach. The goal is nonetheless always to minimize the amount of events being distributed, be this through dynamically moving areas when a server becomes overloaded, or by limiting the scope of a user. Regardless, most of these partitioning schemes make the assumption that the system consists of a centralized cluster of servers, grid or similar. Thus, little or no efforts have been made

to investigate the effects of a decentralized distributed system middleware, which would allow for regions of the game to be migrated based on the physical locality of the users in addition to their virtual locality. We have already seen that it is possible to determine the relative location of the users, based, for example, on an analysis of their RTT [9]. We also know that a common way of dividing virtual environments is by partitioning them into smaller cells or regions. There is also a number of algorithms for balancing the load. In this paper, we will now look at the system we intend to implement, which will permit for the accommodation of physical locality in addition to virtual locality. This migration of objects based on the locality of the majority of users would orthogonally further reduce the load and more importantly the latency. In the following section we will describe our proposed architecture for locating objects.

3 Name Service

In our distributed system, we wish to perform migration for the purpose of off-loading servers, and for minimizing the response time for reacting to events received from clients. When objects are distributed across multiple nodes in a system, access to objects which are not in local memory require special handling. Primarily, it is necessary to locate the node on which a particular object is currently residing. A name service provides an application with this type of functionality. Depending on the application being developed, there are different approaches to implement the name service. For our application, we have a set of six characteristics which impact the approach we decide to use:

1. The server must be able to handle thousands of concurrent users. For these users to have an optimal experience, the latency needs to be as low as possible, a requirement which is important for all interactive applications.

2. The users can be located anywhere in the physical world and virtual world.

3. Depending on the time of day, there is a high probability that a majority of the users will be represented by a specific time zone.

4. In order to handle the load generated by so many concurrent users, it is necessary to divide the world into regions which are spread out across a number of servers.

5. Clients and servers use the same libraries, that means that code is shared and that only data needs to be migrated. It also means that we can call any function of a remote object directly, without having to discover which function interface to use.

6. A large number of objects will be created, with greatly varying life spans. Consider the short life time of the bullets fired by a player's weapon, in contrast to the long life time of the player himself.

Znati and Molka [17] analyzed three approaches to implementing a name service; in form of centralized, distributed and hybrid versions. Prior to contacting the target object itself, the centralized version contacts a name server to obtain the objects location in the network. As such, the centralized naming scheme adds an extra level of indirection to the name resolution process. The distributed paradigm removes this level of indirection by placing the name of the object with the object itself. The hybrid approach is based on the design principle of keeping names together with the objects they are bound to on the local level, but resorts to multicasting when resolving names at a regional level. This study indicates that the choice of model for a name service will influence the performance of the service and the throughput of the network. The results showed that the centralized model could achieve acceptable performance only as long as the ratio of remote to local requests was kept reasonable. The performance of the hybrid model highly depended on the efficiency of the

cache design. With all other network conditions set equal they found that, relative to the response times of the centralized simulation, the response time of the distributed simulation were smaller.

A fundamental problem with the centralized version is that all object resolution and registration is performed at a single point in the system. This means it easily can become a bottleneck in the system, particularly considering how high the object creation frequency can become in such systems. As such, it becomes apparent that a centralized version wont scale very well for systems which experience heavy traffic, which is the case for MMOGs. A centralized version also introduces a single point of failure in the case of a crash. It also raises an interesting question with regards to decentralized systems, such as our middleware, about where to place the name service relative to the servers in the system. The hybrid version of a name service solves a few of these issues, but introduces a few of its own. There is no longer a single point of failure, since the name service is distributed. Thus, only the objects managed by the crashing node will become unavailable. Though it still leaves the issue of partial failures to be handled. A possible weakness is in the way objects are located, the lookup method relies on multicasting, and there is a high probability the response time will be too high, particularly for a decentralized system. Since objects are bound locally, rapid object creation and destruction no longer creates the same problems as with the centralized version. Last there is the distributed approach, which raises an interesting issue; in that there is no clear way to show were an object is located without first contacting its name service and how that name service is located given only a high-level name. For our middleware, this is not an issue since we have a single point of initialization. A single node in the system initiates all communication and object creation, thus there is always a known path to an object. The distributed version also leaves the issue with partial failures unresolved, but apart from this it serves the purpose of our middleware well.

Given the characteristics described in the start of this section, and the choice of our name service model, we will now outline the architecture of our middleware, using terms from Mobile IP [18,19]. Mobile IP addresses the desire to have continuous network connectivity to the Internet irrespective of the physical location of a node. This coincides with our goals in that the objective is to make mobility transparent to the application. The analogy is suitable, because where Mobile IP is used to find a route to a mobile computer, moving from network to network, we need to find a route to a mobile object, moving from computer to computer. A prerequisite to accomplish this, is being able to uniquely name a computer or object in the context it is used. We find that the taxonomy used to describe these processes overlap. In the following discussion, we will primarily focus on the definitions of mobile node, home agent, foreign agent, care-of address and home address.

When a mobile node (object) has migrated to another node in the system, it registers its presence with the foreign agent (name service) at its new location. The foreign agent issues a message to the mobile node's home agent (name service), in the form of a care-of address (local object identifier), which the home agent can use to forward requests to the mobile node. Each node in the system has an active name service. This name service can take on the characteristics of a foreign agent and a home agent. There is, however, a logical difference, depending on whether the object is propagating to or from a node in the system. At this point, our implementation diverges from the approach of Mobile IP, where a mobile node has one home agent throughout its lifetime. As mentioned, a single server is used to initialize the system, objects composing the system are propagated to other servers based on necessity, and links to the propagated objects are maintained in the name service of the corresponding servers. If this were not the case it would be impossible to maintain links between objects, because we would have no way of locating them. This is a result of our distributed name service. For an object to function as a mobile node, it needs to be serializable. Serialization is the act of storing the state of an object with the intention of restoring it again at a later point in time. This functionality is commonly used to move objects between servers, and makes it possible to migrate objects. As such, these serializable objects can be added to migration groups. If and

HOSTNAME	PORT	LOCAL IDENTIFIER		
		OBJECT ID	TIMESTAMP	PSEUDORANDOM NUMBER

Figure 1: Format of the Home Address and Care-of Address

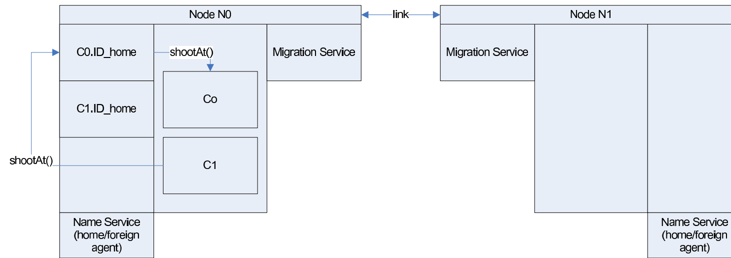


Figure 2: Overview of nodes before migration

when an object is added to a migration group depends on the currently activated migration policy. The migration policy can be preemptive as will be the case for a locality based load-balancing, or reactive if the load on the server becomes to great.

The care-of address and home address, which are used to identify objects in the system, have a format as seen in figure 1. The address uniquely identifies an object in the system throughout the lifetime of the application, since this type of application is designed to run indefinitely. We see that there are three main sections which compose the address. Each is required to identify an object because of our distributed name service. The hostname identifies the node where the object is located. The port provides an access point to the name service, which maintains local objects. The local identifier is specific to an object in the name service. The object-id is an index to more information about the object, such as the pseudo-random number and timestamp. The timestamp is required to identify the object temporally, but since this does not guarantee it to be unique over time, because of uncertainties related to computers and time keeping, we have a pseudo-random number in addition.

In order to solidify the architecture presented here, consider the lifetime of a fictive object in our distributed environment. In this scenario, we have two nodes in our system, the initializing node, N_0, and the secondary node, N_1. At startup, all collaborating services are initiated, such as the name service, migration service, host service and similar. After initialization, a player connects to the application, and an object C_0 representing his character is instantiated at N_0. Since C_0 is a serializable object, an identifier is generated called $C_0.id_{home}$ which is logically equivalent to this object's home address. The format is as detailed in figure 1. N_0 has in effect become the home agent of C_0. C_0 now exhibits all the properties of a mobile node: it is possible to migrate it, and it has a home agent. A short while after, a second player connects to the server, and the object C_1 is created. Figure 2 shows what the node would look like at this point in time. If these two players are fighting each other, and C_1 decides to shoot at C_0, we can see that the *shootAt()* function call goes via the name service, which determines that C_0 is a local object and directs the shot accordingly. As time goes by, more users connect to N_0, and eventually a migration policy is activated. Based on the requirements of the migration policy, C_0 is added to a migration group. Based on an analysis of available nodes in the system, a node is selected as the recipient of the objects. Depending on the the requirements of the migration policy, we might want a node as physically close as possible to the majority of users. Based on the requirements issued by the migration policy, N_1 is selected by the host service as the destination node. At this point, C_0 is migrated to N_1 (see figure 3). The name service at N_1, which is logically equivalent to a foreign agent, accepts the mobile node C_0 and generates an identifier, $C_0.id_{care-of}$, which is logically equivalent to its care-of address. This identifier is sent in return to the home agent and replaces the home address $C_0.id_{home}$. If the second character C_1 now fires a shot at C_0, the *shootAt()* function call will go via the name service, but

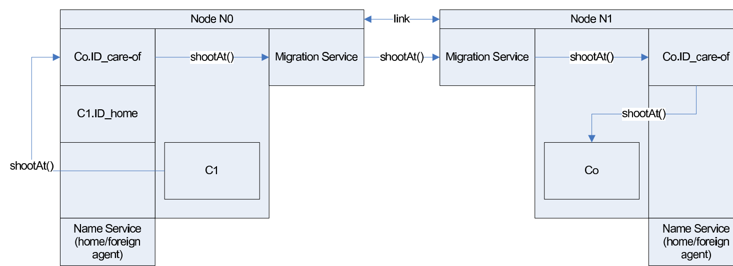


Figure 3: Overview of nodes after migration

unlike last time, C_0 is no longer a local object. Instead of accessing the object locally, the care-of address is used to issue a remote procedure call to the object at its new location.

Following this example a few things become apparent. Any serializable object is identified through the name service by querying on its identifier, i.e., an indirection which is necessary for accessing objects at remote nodes. It also becomes apparent that we are highly vulnerable to partial failures. This happens when a node in a distributed system becomes unavailable, effectively rendering the objects managed by it inaccessible.

While the current architecture does not accommodate for partial failures, there are ways to minimize the repercussions of these incidents. One possibility is to utilize a central registry, where all object migration is recorded. In the case of a miss, when attempting to access a remote object, the central register can be queried instead. Once the node has recovered, normal object access resumes. A flaw with this approach, is that the central register becomes a single point of failure in the case of a crash. In addition, it must be capable of handling the traffic generated by all migrations, including any misses. This is a poor sign, since migration most likely is activated as a result of heavy load. A different approach has its roots in peer to peer based filesystems, where copies of an object will be distributed to several nodes in the system. PAST [20] and OceanStore [21] have, for example, implemented such systems with success. PAST copies objects to random nodes, in an attempt to distribute the objects evenly. OceanStore uses a more deterministic approach, and places the objects close to nodes which access them. Lookup of objects in the system can then be implemented in a fashion similar to that of Chord [22] or Tapestry [23]. These implementations are based on the principle of incrementally forwarding messages from point to point, until they reach their destination. Each node in the system keeps a small routing map, which is used to determine which nodes to forward the message to. A problem with this type of lookup is that the response time might be too high for interactive applications. Both the centralized and distributed fail safe techniques offer their own set of advantages and disadvantages. As such, we intend to investigate viable methods for recovering from partial failures in our middleware in future work. In the following section we will look at the process used for automating the creation of serializable objects.

4 Code Generation

Most computer games are developed using the object oriented programming language C++ [24]. Consequently, our middleware is also implemented using this language. Since C++ has no built-in mechanisms for the serialization of objects, and manually writing code for this is a tedious and error-prone process, we provide the application developer with a tool for automatically generating this functionality. The result of the generation process is a skeleton, which can easily be integrated with the middleware. We implement the serialization mechanisms using inheritance, polymorphism, run-time type identification (RTTI) and virtual functions.

In order for an object to be serializable, it must inherit the abstract class called *Object*. This class defines a set of pure virtual functions, which all derived classes need to provide an implementation for. These functions perform the serialization and deserialization of an object. When objects

are passed to the migration service and migrated, they are up-cast to look like an instance of an *Object*. Since the serialization functions are implemented as virtual functions, the derived object's implementations of the functions are called. The type name of the object always precedes the object itself on the stream. When deserializing the object at the receiving end, the correct deserialization function is called for the object by looking up the type name of the class in a type register. The type register consists of mappings between the type name and an instance of the corresponding class for all serializable classes. The type name of a class is determined by a python script during generation and is a combination of the filename and class name, this is done because RTTI type names are not portable. Once the object is deserialized it can be downcast accordingly, using RTTI, to an instance of its type, so the functionality associated with such an object becomes available. The reason it needs to be downcast is because the deserialization process returns a reference to an instance of *Object*.

Serialization in our system is necessary so we can transmit objects across the network. As a side effect of this, we need an architecture and operating system independent encoding. The rpcgen [25] utility was developed for generating client and server stubs for remote procedure calls, we use rpcgen to automatically generate routines for serialization using C-like data structures. Furthermore, rpcgen uses the eXternal Data Representation (XDR) [26, 27] format for the serialization of parameters, which is a standard for the description and encoding of data. It is useful for transferring data across networks between different computer architectures. XDR is based upon implicit typing. The sender and receiver must agree on the order and type of all data. Moreover, XDR makes use of symmetric data conversion. Both the client and server convert from and to a standard representation. XDR routines are direction independent, the same routines are called to serialize and deserialize data. XDR supports all C data types, such as int, double, char, and arrays of these types. Filters are provided for the serializing and deserializing to and from their local representation. These basic filters can be combined to allow for more complex data types, such as structures, to be serialized.

Figure 4: Code generation process

Figure 4 depicts the process used to generate the skeleton. It is implemented as a python script. To parse the C++ header files, we make use of the GCC-XML [28] parser, which is a tool that extends the open source GCC compiler, using its internal representation to produce XML output. Based on information obtained by processing the XML-file, we derive rpcgen language structure definitions. These definitions are in turn processed by the rpcgen tool which generates functions for serialization. The output of the process is a skeleton we have created, which combines the components of the structure described earlier in this section. The skeleton generator expects a C++ class declaration as its input. In addition to normal C++ class syntax, GCC-XML allows for defining additional attributes. We make use of this ability to extend the C++ syntax with our own keywords.

When an object is migrated, one does not necessarily want all the data to be serialized. We therefore provide a special keyword (*_serialize*, see figure 4) to specify what data is to be serialized. Notice how the *counter* variable is not serialized, in contrast to the *damage* variable. Any number of these keywords can be added to aid in the parsing process. Other suitable keywords will be introduced to support remote method invocation, mark the classes that can be migrated and so forth.

5 Conclusion

MMOGs are distributed applications that have a range of unique characteristics, e.g., the stringent requirement to latency, which means that to cope with the server load the virtual environment needs to be logically divided across a number of servers. Another factor is the diversity of locations among the users connected to the application, and the fact that users located in the same virtual region, will be connected to the same physical server, and as such cannot necessarily be separated.

The assumption is, that based on an analysis of the RTTs for users in a given virtual locality on a server or by looking at IP addresses if available, we can identify if the majority of those users are located at the same geographical location in the world. Our assumption is based on the fact that people from certain time zones will be more active depending on the time of day. Given this, we believe that the overall latency of that area can be lowered by migrating the region to a location as close as possible to the majority of the users. It is our understanding that little or no work has been done on how latency can be improved based on this type of migration policy. It is not to say that this is the only type of migration policy that needs to be present on the server. Migration can also be triggered because the server is becoming overloaded.

In this paper, we have therefore presented a work in progress for a middleware which accommodates the physical locality of the users in addition to their virtual locality. This is accomplished through our object model, which takes advantage of polymorphism and virtual functions, to make C++ objects serializable, and thus transferable. In order to distribute and locate objects across multiple servers, in an efficient manner, we use a distributed name service. Additionally, our middleware provides the developer with a tool for automatically generating skeletons, in order to minimize the number of coding errors and ease the development of the application. We intend to implement and test the feasibility of this architecture for use with a real MMOG, and also to see if there are any real benefits of migrating based on locality. In order to accomplish these goals, the middleware will eventually need to be extended with functionality for handling concurrency control. Susceptibility for partial failures and the possibilities for recovering from such incidents also needs to be examined.

Finally, we must add functionality for selecting which objects to migrate, e.g., migrating all members maintained in a group for group communication. This is ongoing work [29], and the intention is to eventually combine these components, and analyze the results.

References

[1] The Entertainment Software Association. ESAs 2006 essential facts about the computer and video game industry. *http://www.theesa.com/*, jan 2007.

[2] B. S. Woodcock. An analysis of mmog subscription growth. *http://www.mmogchart.com*, jan 2007.

[3] M. Claypool. The effect of latency on user performance in Real-Time Strategy games. *Computer Networks*, 49(1):52–70, 2005.

[4] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. *Proceedings NetGames' 04, Portland, Oregon, USA*, pages 144–151, aug 2004.

[5] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. *Proceedings of NetGames' 05, Hawthorne, NY, USA*, pages 1–7, oct 2005.

[6] Electronic Arts. Ultima Online. *http://www.uo. com/, January*, 2007.

[7] CCP. EVE Online. *http://www.eve-online.com/, January*, 2007.

[8] Blizzard. World of Warcraft. *http://www.worldofwarcraft. com/, January*, 2007.

[9] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. *Proceedings of NOSS-DAV' 06, Newport, RI, USA*, pages 1–7, may 2006.

[10] Funcom. Anarchy Online. *http://www.anarchy-online.com/, January*, 2007.

[11] Abdennour El Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer mmog architecture. *Comput. Entertain.*, 3(2):3–3, 2005.

[12] Linden Lab. Second Life. *http://secondlife.com/, January*, 2007.

[13] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. *Proceedings of NetGames' 05, Hawthorne, NY, USA*, pages 1–7, October 2005.

[14] T.N.B. Duong and S. Zhou. A dynamic load sharing algorithm for massively multiplayer online games. *ICON' 03, Sydney, Australia*, pages 131–136, October 2003.

[15] Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. Matrix: Adaptive middleware for distributed multiplayer games. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 390–400. Springer, 2005.

[16] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A Distributed Architecture for Online Multiplayer Games. *Proceedings of NSDI' 06, San Jose, USA*, pages 155–168, May 2006.

[17] T. B. Znati and J. Molka. A simulation based analysis of naming schemes for distributed systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 42–53, Los Alamitos, CA, USA, April 1992.

[18] C. E. Perkins. Mobile IP. *Communications Magazine, IEEE*, 35(5):84–99, 1997.

[19] C. Perkins. RFC 2002: IP mobility support, October 1996.

[20] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of SOSP'01, Lake Louise, Alberta, Canada*, pages 188–201, oct 2001.

[21] Dennis Geels. Data Replication in OceanStore. Technical Report UCB//CSD-02-1217, Computer Science Division, U. C. Berkeley, nov 2002.

[22] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of SIGCOMM' 01, San Diego, CA, USA*, pages 149–160, aug 2001.

[23] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, apr 2001.

[24] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[25] Sun Microsystems. *rpcgen Programming Guide*. Sun Microsystems Inc., Mountain View, CA, 1987.

[26] Raj Srinivasan. XDR: External data representation standard. RFC 1832, August 1995.

[27] Sun Microsystems, Inc. XDR: External data representation standard. RFC 1014, June 1987.

[28] B. King. GCC-XML the xml output extension to gcc. *Undated. Online: http://www. gccxml. org/HTML/Index. html.*

[29] K. Vik, C. Griwodz, and P. Halvorsen. Applicability of group communication for increased scalability in mmogs. *Proceedings of NetGames'06, Singapore*, oct 2006.

Paper II

Latency Reduction by Dynamic Core Selection and Partial Migration of Game State

Paul B. Beskow, Knut-Helge Vik, Carsten Griwodzand Pål Halvorsen

Published: Proceedings of Workshop on Network and Systems Support for Games (NetGames), ed. by Mark Claypool, pp. 79-84, ACM (ISBN: 978-1-60558-132-3), 2008

Evaluation: 31 papers submitted, 15 accepted, with acceptance rate of 48.4%

Author contribution: Beskow was the driving force behind this article and acted as the primary investigator. Vik contributed with input on the choice of core-selection algorithm.

Latency Reduction by Dynamic Core Selection and Partial Migration of Game State

Paul B. Beskow, Knut-Helge Vik, Carsten Griwodz, Pål Halvorsen
Department of Informatics, University of Oslo, Norway   Simula Research Laboratory, Norway
Email: {paulbb, knuthelv, griff, paalh}@ifi.uio.no

ABSTRACT

Massively multi-player online games force developers to deal with the conflicting requirements of supporting large numbers of concurrent players, while simultaneously maintaining low latency. As a result, a common way of distributing load is by dividing the virtual environment into virtual regions. We propose the use of core selection for finding an optimal server for placing a region and support for migrating the game state to that server. The first goal relies on a set of servers and measurement of the interacting players latencies. By locating an optimal server, we wish to decrease the overall latency of the majority of players. This reduction occurs by migrating the region to a server closer in proximity to the majority of players in that virtual region, thereby lowering the response time of any interaction. The geographical proximity of players can be determined by analyzing IP addresses, RTTs or similar.

1. INTRODUCTION

"Lagger!" is a likely expression to hear uttered in a real-time interactive online game. This term addresses players with excessive latency, which in the gaming community is colloquially referred to as *lag*. It should be understood that this term holds no positive connotation, as a player that has high latency will inadvertently have a negative effect on the perceived quality of the game play [1,2]. This occurs, as most online games are based on a client-server model, where events are collected at the server, and in turn distributed to the interacting players. By its nature, if one player is on a slow connection, any added delay is not isolated to the player alone, but will propagate to other interacting parties, potentially resulting in inconsistencies. While having minor effects on the outcome of the game, it results in a perceived deterioration to the quality of interaction [3]. As such, *low latency* for *all players* is a prevalent goal.

As latency is affected by physical distance between the player and the server, reducing the latency to all players is an arduous task. Thus, providing the required level of interactivity, within the latency requirements of the game (ranging from 100 to 1000 ms [4]), becomes very hard and may demand that the player is in *close proximity to the server*.

As an example, consider massively multi-player online games (MMOGs), which are persistent online worlds that allow thousands of users to interact concurrently in a virtual environment. To support this many concurrently interacting players, the virtual environment is commonly split into virtual regions. This makes it possible to distribute the regions across a number of (possibly geographically distributed) servers, this as the regions are logically decomposed, i.e., each, in effect, responsible for handling some regions and the players interacting in each region. As the geographical dispersion of players in MMOGs depends heavily on the time of day [5], it is possible to have players clustered by their virtual (belonging to the same region) as well as physical location, e.g., if most players are from Europe, then use a server in Europe. Therefore, since a user's proximity to the server impacts the latency, and in turn, since latency is an integral factor for the playability of an online game, this raises an interesting question: *How can we optimize the placement of a virtual region and its interacting players given a set of globally distributed servers?*

In the remainder of this paper, we will describe our use of core selection for finding the most appropriate server for placing a virtual region, when given a player base and a set of available servers. Once a server has been selected, we use our migration functionality to move the active region to its new location. To maintain an optimized set of references to the migrated game state we use a distributed name service. On the background of existing work in distributed systems, we believe that with the combination of *core selection, a distributed name server and migration* we have a viable solution for creating a globally distributed game, which is capable of *lowering the overall latency of the interacting players.*

In section 2, we look at the basis for some of our assumptions, and take a closer look at related work. In section 3, the core selection process is described in detail and how it can be applied to the scenario we have described in a reasonable way. In section 4, we look at the migration functionality and how it is supported by the distributed name service. In section 5, we evaluate the core selection process and migration functionality. Finally, we summarize our findings in section 6.

2. BACKGROUND AND RELATED WORK

2.1 Game Characteristics

The body of work that analyzes game traffic has grown considerably in the recent past. The main conclusions in our scenario are that 1) game traffic varies strongly with time and the attractiveness of the individual game [6, 7], 2) some latency is tolerable [1], as long as it does not exceed the threshold for playability, i.e., ranging from 100ms to 1000ms depending on the type of game [4] and 3)

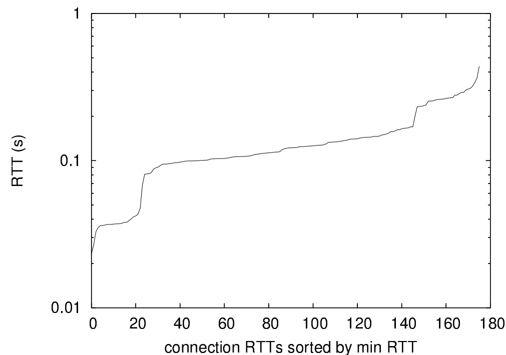


Figure 1: AO: connection RTTs sorted by min RTT

geographical dispersion of players in an online game depends heavily on the time of day [5]. In addition to these works, we have analyzed packet traces (see [8]) from Funcom's popular role-playing MMOG *Anarchy Online* (AO) [9]. Statistics from the traces reveal that there is a potential gain with respect to latency by using our migration middleware. For example, in one of the game regions within a timespan of about one hour, we found approximately 175 distinct connections. These are sorted according to their measured RTT in figure 1. With the knowledge that the servers are located in the US, the observed minimum latencies in the figures indicate that there are players concurrently located in the US, Europe and Asia. The number of players in different areas of the world also typically vary according to the time of day, and finding an appropriate location for the server might be of vital importance in order to meet the latency requirements, i.e., as the majority of users are in the second range in figure 1, the average latency could be reduced by moving the game region to a server in Europe.

2.2 Server selection

Game server selection is an important facet for the playability of a game. This is particularly true for games that are highly sensitive to latency, such as first person shooter (FPS) games. As such, the player's selection process is commonly guided by measuring dimensions that affect playability, such as latency and packet loss. This sensitivity to latency is commonly alleviated by having a high distribution, and availability, of servers. To this respect, Chambers et al [10] have looked at how server selection can be optimized for a single client given a set of available servers. Following this scenario and figure 2, where the circles denote acceptable boundaries for playability, we see that the servers in *Chicago* and *New York* will be weighted equally for *Jane*, while only *New York* is acceptable for *Anette*. In a further study, Claypool [11] notes that we regularly find groups of players that wish to play together on a server, such as friends or clans (organized players). As such, he has investigated how server selection can be optimized from the perspective of a group of players. Given this scenario and figure 2, the guided selection process would weight *Chicago* higher then *New York* fpr *George*, *Jane* and *Tom*, while *New York* is weighted the highest for *Jane* and *Anette*. A third study, which is related to these topics, looks at how the search itself can be performed in an optimal way [12]. The two former papers both have in common that they consider server selection from the perspective of the player(s), and additionally assume a certain availability of servers. As we can see from figure 2, it is common for geographically coupled users to play against each other. For a world spanning game, where all users interact in the same game instance, such as an MMOG, there is often a limited number of servers to select from. Thus, the differences in geographical locality become more apparent. When we consider the time-of-day characteristics of geographical dispersion, it would be beneficial to examine if these techniques can be applied by a server, to improve the playability for a group of players, e.g., the players in the same virtual region.

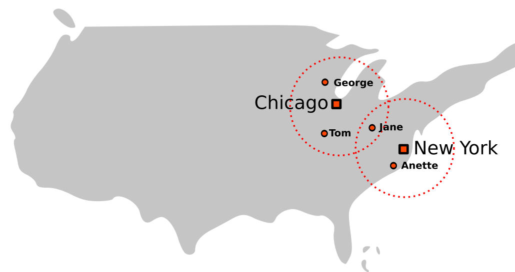


Figure 2: Clustering in FPS style games.

2.3 Virtual regions

As MMOGs are expected to handle thousands of concurrently interacting players it is common practice to use a static, region based partitioning scheme. The virtual environment is thus divided into smaller, more manageable parts, where each virtual region is hosted on a single server in the cluster. A widely accepted problem with the static partitioning scheme is that it does not take into account the dynamic nature of MMOGs. Even if the static partitioning is based on population density trends, and arranged to accommodate this, it is still susceptible to imbalances due to unforeseen events. Thus, a lot of research focuses on how to improve the flexibility of these partitioning schemes, and consequently, algorithms for efficiently distributing entities and regions. This research, however, does not address how the locality of the users, in relation to a server, will effect latency. For example, Turck et al [13] have investigated the effects of dividing a game world into dynamic micro-cells. A study with a similar background is performed by Duong et al [14]. Such micro-cells can be reassigned to servers in a cluster if the load on the server they are currently residing on becomes too large. Different load-balancing algorithms were applied, none of which factored in locality of users, and the number of micro-cells supported per server was varied. The test was performed on a centralized cluster. A similar approach is deployed by IBM with the Matrix [15] middleware. The focus is on the player, where consistency updates are limited to an *area of interest*. It is based on the observation that MMOGs are nearly decomposable systems, and as such, it is usually sufficient to update players with only those events that occur in their zone of visibility. Matrix makes use of region based partitioning as an underlying foundation, but this is for the purpose of easily distributing the virtual world across multiple servers. Consistent for all these approaches is that a dynamic solution outperforms a static one.

2.4 Summary

In summary, the work on static and dynamic partitioning consider server load in a centralized cluster and not latency due to the geographical location of users, this despite the fact that there is visible evidence of such a trend. Most of the research tries to optimize the partitioning of the virtual environment into regions which can dynamically accommodate hot-spots. Furthermore, with respect to server selection, previous work look at how to best select the best game instance (machine), e.g., with lowest latency. However, in these tests, little or no efforts have been made to investigate the effects of a decentralized distributed system middleware, which would allow for regions of the game to be migrated based on the

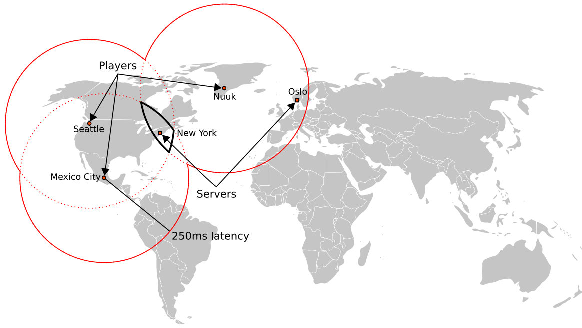


Figure 3: Latency as physical distance

physical locality of the users, in addition to their virtual locality. Our analysis of the game traffic shows that the there are players connected from all around the world (see figure 1), but the amount of players from a part of the world will depend on the time of day. Thus, as there is a shift in the location of the majority of players, another approach to reducing the latency, both due to RTT and loss, is to dynamically find the center of the group of players and migrate the game objects to a server whose location is closer to the majority of the users. We can accomplish this through core selection, which helps us determine which node to migrate the players to.

3. CORE SELECTION

An efficient means to reducing latency can be accomplished by migrating game state to an appropriate server. This appropriate server may be a server close to the center of a given group of players. As an example, consider an MMOG with geographically distributed servers (see figure 3). At any time, a server can be hosting none, some or all of the virtual regions making up a virtual environment of the same game instance. Given a region, there are currently a number of interacting players, and as we have seen earlier, there is a high probability that the majority of these players are in reasonable proximity of each other, as seen from a geographical perspective at a given time. Furthermore, looking at figure 3, we can see how a latency requirement can be translated into a measurement of physical distance. In this example, we have two server nodes, one located in *New York* and one in *Oslo*. In addition, we have three players that are connecting from *Mexico City*, *Seattle* and *Nuuk*. In case of a 250 ms delay requirement from the server (500 ms pairwise latency in RTS), the bordered intersection indicates an optimal area for the server to be located, and given the set of available servers, the natural choice would be to select the server in *New York* as the core server node.

Accordingly, it is desirable to determine if the server currently hosting a region is the optimal choice. In this context, optimal will be determined by the overall latency experienced by the players currently interacting there. As such, we wish to ascertain whether there is a server in the system that would be able to provide these players with better overall performance in terms of network delay. Looking at figure 3, with server nodes in *Oslo* and *New York*, we can naturally assume that the core selection process would have selected *New York* as the optimal location.

To accomplish this, we can use core selection techniques, which in this scenario requires complete information about the available server nodes, and players interacting in the region. Based on this information, the heuristic core selection method will determine which server provides the optimal placement for that region and its players.

3.1 Core selection heuristics

The core search is conducted by core selection heuristics that are devised from a graph theory perspective. Upon core selection, the core nodes may be used to administrate clients that join and leave groups. Such groups can be defined and updated dynamically in an MMOG, for example, based on some area-of-interest management like the same virtual region. Names given to core nodes include leaders, core nodes or rendezvous-points. Typically, the core node of a group is contacted for each membership change, such that it always has the latest view. When a limited set of nodes handles the membership management, it simplifies membership updates, and applications that have highly dynamic groups require fast and simple group management.

Core-based protocols work on the assumption that one or more core nodes are selected as group management and forwarding nodes. Therefore, the cores need to be selected using some core selection heuristic. Several core selection heuristics have been proposed, and a comprehensive study is given by Karaman and Hassanein [16]. An overall goal is to select cores on the basis of certain node properties, such as, bandwidth and computational power. We wish to base this decision primarily on latency. The cores that are selected depend on the group size and location, as well as the capacities in the available core nodes. In this paper, our focus is on electing a single core node for each defined group. The core may, for example, be a server or a proxy administrated by the game provider.

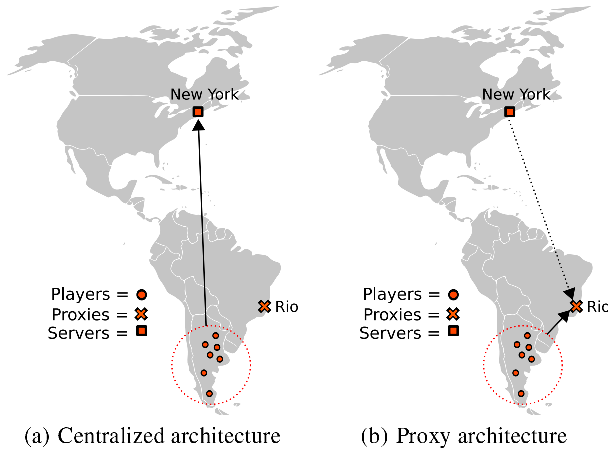
3.2 Core selection in a proxy architecture

Today's typical client/server model makes it easy to manage the global game state, but it has drawbacks. The server is a potential bottleneck, both in terms of computing and bandwidth capacity, and the latency heavily depends on the physical distance from each individual client to the server. In figure 4(a), we illustrate an example where a centralized server stores the game state and cannot take into account the physical location of the clients.

Proxy technology and peer-to-peer are distributed options. Proxy technology has an infrastructure with a centralized server and a set of distributed proxy servers. Some proxies are usually physically closer to the clients than a central server, and may, for example, hold partial game state copies of the virtual game regions needed by the clients connected to it. The proxies can be organized hierarchically, each responsible for a fixed set of clients based on location, or, in a proxy pool fashion, where clients connect to the proxies that are best suited in a given situation. In figure 4(b), the central server has migrated the state to a proxy that is closer to the group of clients. A peer-to-peer architecture distributes the game state among peers and have no central server, making it very hard to administrate the game state such that it is consistent. Currently, MMOGs can not solely use a peer-to-peer architecture because there is no working business model.

In our work, we focus on proxy technology because it allows a trade-off between client/server and peer-to-peer advantages and disadvantages. However, a mix of client/server and peer-to-peer communication styles may be used for different traffic types fitting to these models.

The core selection heuristics presented here search among a predefined set of servers and proxies to find one optimal core, which is the *graph median*. The graph median is the node for which the sum of lengths of shortest paths to all other vertices's is the smallest. The heuristics are [16]:



(a) Centralized architecture (b) Proxy architecture

Figure 4: Architecture

- *Topology Center:* Find a central entity (server) that is closest to the topological center of the *global graph*.
- *Group Center:* Find a proxy that is closest to the group center of the *group graph*.

The topology center heuristic is given as input a set of available servers that are located around the world. The heuristic searches for the server for which the sum of latencies of shortest paths to all clients in its member network is the smallest. The group center heuristic similarly searches among a set of available proxies located around the world. It is given as input a group of clients, and based on this the group center heuristic selects the core proxy to be the proxy for which the sum of latencies of shortest paths to all the clients in the group is smallest. These simple heuristics form powerful techniques in the search for suitable hosts to migrate game state to.

4. MOVING WORLDS WITH MIGRATION

The core selection process is responsible for determining whether the current server hosting a virtual region (based on its player population) is optimal. In the case where it is able to locate a more appropriate server, the MMOG will move the game state of that region to its new location. To accommodate this process, we have developed a middleware that is capable of performing such migration of game state, which consists of a number of interacting objects (following the object-oriented paradigm). Before migrating an object, we must know that all references to that object will be maintained. To accomplish this, we use a name service. This service is responsible for keeping an up-to-date index for the location of objects in the distributed system, and redirect method invocations accordingly. As such, the reduction in latency is accomplished by decreasing the response time of, for instance, remote method invocations (RMI). This because we move objects closer to the majority of the players.

4.1 A name service

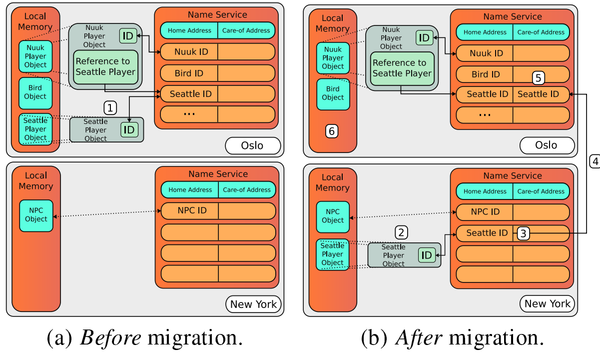
The name service maintains references to the objects in the distributed system, this goal can be accomplished in several ways. Znati and Molka [17] analyzed three approaches to implementing a name service; in the form of centralized, hybrid and distributed versions. Prior to contacting the target object itself, the centralized version contacts a name service, located at a well-known server, to obtain the objects location in the network. As such, the centralized naming scheme adds an extra level of indirection to the name resolution process. The hybrid approach is based on the design principle of keeping names together with the objects they are bound to on the local level, but resorts to multicasting when resolving names at a regional level. The distributed paradigm removes this level of indirection by placing the name of the object with the object itself. Once an object resolution has been performed, the object is accessed directly at the server managing the object. The results showed that the centralized model could achieve acceptable performance only as long as the ratio of remote to local requests was kept reasonable. The performance of the hybrid model highly depended on the efficiency of the cache design. With all other network conditions set equal they found that, relative to the response times of the centralized simulation, the response time of the distributed simulation were smaller. We relate these three models to MMOGs with geographically distributed servers by coupling them with the following characteristics:

1. There are thousands of concurrently interacting players.
2. The virtual environment is divided into virtual regions.
3. Players are dispersed physically as well as virtually.
4. The physical player distribution depends on time of day.
5. The servers in the system are geographically distributed.
6. Code is shared so only data is migrated.
7. There occurs frequent object creation and destruction.
8. Efficiency is more important then consistency.

Given these characteristics a distributed name service best suits our needs, primarily because efficiency is more important then consistency in this scenario. It is more efficient because there is no overhead in binding an object with the name service, as this occurs locally. Given that the servers in the system are geographically distributed this becomes essential. Other advantages are that there is no single point of failure, which implies that large parts of the application can continue running if a server were to fail. Looking up objects will also be efficient, as we can directly query the node our name service has registered as the current maintainer. Though it is worth nothing that this access time will depend on how many times an object has been migrated (from its point of creation) and at which point in this chain the invocation is performed. For further details and a thorough discussion about the implementation of the name service, references and migration see [18]. In the following section we will solidify our understanding of the described mechanisms by looking at an example of them in use.

4.2 In action

Consider a system consisting of two servers, as seen in figure 3. As it is currently evening in Europe, the majority of the interacting players are European. As such, all of the virtual regions are currently being hosted in *Oslo*. In addition to the European players, we have a couple of players connected from *Nuuk* and *Seattle*. Looking at figure 5(a) we see that these two players have been bound to the name service and received an identifier (the other players are not shown in this example). We can also see that the *Nuuk* player references the player in *Seattle*. Notice that this reference is not to



(a) *Before* migration. (b) *After* migration.

Figure 5: Server configuration

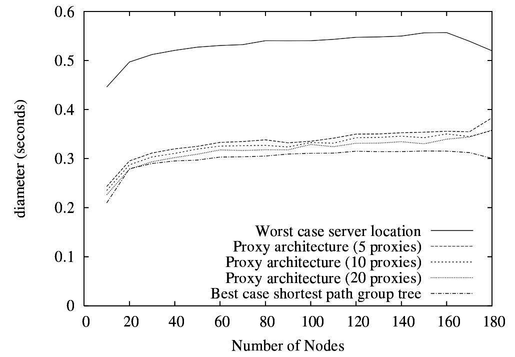


Figure 6: Diameter of group trees using a proxy architecture

Invocation type	# of Invocations	Mean
Remote	100,000	2.26541 *milliseconds*
Normal	100,000	0.10125 *microseconds*

Table 1: Cost of method invocation.

the player object in local memory, as we might expect, but to the identifier (created earlier) in the name service. As the time passes the population on the server shifts from being predominantly European to American. As such, the core selection process is performed for each virtual region. For some of the regions it is found that the server in *New York* is best fit to serve the currently connected players. As we can see from figure 5(a), the region with the players connected from *Nuuk* and *Seattle* is marked for migration. At this point in time, the region these two players are interacting in is migrated to the server in *New York*. We will see how this is accomplished in our middleware, when we follow the steps outlined in figure 5(a), which denotes the actions before migration, and figure 5(b), which denotes the actions taken during and after migration.

The *first* step (1), consists of serializing the player object. Serialization is the act of creating a binary representation of the object that can be transmitted across the network. In this case, the player object is migrated to our server in New York. The *second* step (2) consists of recreating the object at the receiving side, which is also known as deserialization. Once the object has been recreated, it is bound with the local name service, which is the *third* step (3). At this point in time, the Oslo server knows that the player object is in New York (it sent it there after all), but will have no way of forwarding requests, as it will have no way of knowing what object in New York to forward requests to. This is because the player object has been given a new identifier in New York, and, in effect, has no knowledge of its past history from the server in Oslo. This is why we in the *fourth* step (4) need to return the identifier assigned to the player object by the server in New York. After Oslo has updated its name service by associating the received identifier with the player object, which happens in step *five* (5), we can without worry remove the object from local memory; as is done in step *six* (6). The reason for this is that any invocations made to the player object will now be intercepted correctly, because the name service in Oslo now knows the identity of the object in New York.

5. EVALUATION AND DISCUSSION

We implemented our core (proxy) selection heuristics in a simulator that mimics group communication in a game using a preselected central entity to handle the membership management. The central entity is always selected using the core selection heuristic topology center. Furthermore, we generated network topologies with the BRITE Internet topology generator [19], and in our experiments, we used flat, undirected Waxman topologies [20]. From this network, we created a fully connected undirected mesh, representing application layer communication that is shortest-path routed. The network size was 1000 nodes, and all the nodes join and leave groups throughout the simulation, causing group membership to be dynamic. Group popularity was distributed according to a Zipf distribution [21]. The network layout is a square world with sides equal to 100 ms.

Our goal is to illustrate the significance of server location. In addition, we want to show that having a proxy architecture with a limited amount of proxies can dramatically reduce the overall latencies. We can express the worst-case pair-wise latency between clients in a network through the diameter (measured from, f.ex a core node), and it is desirable that the diameter is as low as possible. Figure 6 shows the diameter of shortest path trees for different group sizes where servers and proxies are the roots of the trees. We can see that having a limited amount of proxies placed around the world can reduce the diameter dramatically. And as expected, increasing the number of proxies will incrementally decrease the diameter. The worst-case placement of a server is not an invalid situation, because if a single server administrates the entire world the worst-case placement of a server will happen for a large number of the clients as the day passes.

As our results show, we are able to find suitable proxies which will lower the diameter of the distribution tree.

The described middleware has been implemented and tested as a proof of concept. We tested the middleware by implementing a basic protocol for group communication, with the intent of imitating the interaction we would expect to see in a virtual region. Running two servers, we had several clients connect to one of the active servers, where they initiated their communication. As the clients were interacting we migrated the region to the second server, where the clients were able to continue their interaction unhindered. As such, the middleware has shown that it is capable of migrating objects, maintain references to these objects, and reconnect the interacting clients so they can continue their interaction.

To measure overhead, we have timed a remote method invocation and compared it to a regular method invocation. We ran the tests

on an Intel Core 2 Duo, using only one core, which was clocked at 800MHz. The operating system running Linux kernel version 2.6.22-14. During these tests the *sender* and *receiver* were both running on the same machine. The result is summarized in table 1. These tests represent the expected overhead added by the middleware itself for invoking methods on a migrated object, which involves name service look up, serialization, deserialization, and some additional operations. The test methods took *void* arguments and returned an *int*. The overhead of a remote invocation is considerable when compared to a regular invocation, but is negligible when compared to the overhead added by the network. With respect to the latency gain, this is totally dependent on the core node which is found. The results of these tests are reported above.

We also need to consider the overhead of migrating a virtual region, though this will greatly depend on the number of objects being migrated, and the size of the objects in question. Though this can be accomplished transparently, without the player ever noticing. With our middleware we migrate data only, as the code is shared. We give the application developer the possibility of defining what parts of an object they wish to migrate. The serialization mechanisms and more are described in further detail in [18].

6. CONCLUSION

It has been shown that there is a strong correlation between latency and the playability of an online game [4], with the perceived game play deteriorating considerably as the latency increases. An important factor for world spanning games, such as MMOGs, lies in the diversity of its user base. There will, at any point in time, be a number of players connected from different physical locations. It has, however, been shown that distinct groupings will appear, and change with the time-of-day [5, 8]. There have been made few efforts into determining how best to support the dynamic player masses in virtual worlds hosting thousands of concurrently interacting players, when geographically distributed servers are available.

In this paper, we have presented a viable solution, given a world spanning game, such as an MMOG, with geographically distributed servers, as to how this can be accomplished. We have shown how core selection can be used to find an optimal node in the system for placing a virtual region, and correspondingly the players interacting in that region. Once an optimal node has been located we can migrate the game state to that node, maintaining references to the migrated state through the use of our distributed name service. By performing this migration, the overall latency of that region can be lowered, i.e., the decrease in average network latency will by far outweigh the increased overhead of remote method invocations.

Thus far, we have implemented the migration and name service as a proof-of-concept, and run some basic tests to determine its usability. Furthermore, we have run simulations on core selection and determined its applicability in the scenarios we have described. Now, it remains to integrate this functionality with an application and run large-scale tests.

7. REFERENCES

[1] M. Claypool. The effect of latency on user performance in real-time strategy games. *Elsevier Computer Networks*, 49(1):52–70, September 2005.

[2] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. *In the Proceedings of NetGames'04, Portland, Oregon, USA*, pages 144–151, August 2004.

[3] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. *In the Proceedings of NetGames'05, Hawthorne, NY, USA*, pages 1–7, October 2005.

[4] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, November 2005.

[5] W. Feng and W. Feng. On the geographic distribution of on-line game servers and players. *In the Proceedings of NetGames'03, Redwood City, California, USA*, pages 173–179, May 2003.

[6] W. Feng W. Feng, F. Chang and J. Walpole. Provisioning on-line games: a traffic analysis of a busy Counter-strike server. *In the Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement, Marseille, France*, pages 151–156, November 2002.

[7] S. Sahu C. Chambers, W. Feng and D. Saha. Measurement-based characterization of a collection of on-line games. *In the Proceedings of the 5th ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA*, pages 1–14, October 2005.

[8] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 1–7. ACM Press, May 2006.

[9] Funcom. Anarchy Online. *http://www.anarchy-online.com/, June*, 2008.

[10] W. Feng C. Chambers, W. Feng and D. Saha. A geographic redirection service for on-line games. *In Proceedings of the eleventh ACM international conference on Multimedia, Berkeley, CA, USA*, pages 227–230, November 2003.

[11] M. Claypool. Network characteristics for server selection in online games. *In Proceedings of the fifteenth Annual Multimedia Computing and Networking (MMCN'08), San Jose, CA, USA*, 6818:681808, January 2008.

[12] G. Armitage. Optimising online fps game server discovery through clustering servers by origin autonomous system. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, May 2008.

[13] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. *In the Proceedings of NetGames' 05, Hawthorne, NY, USA*, pages 1–7, October 2005.

[14] T.N.B. Duong and S. Zhou. A dynamic load sharing algorithm for massively multiplayer online games. *In the Proceedings of ICON'03, Sydney, Australia*, pages 131–136, October 2003.

[15] Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. Matrix: Adaptive middleware for distributed multiplayer games. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 390–400. Springer, 2005.

[16] Ayse Karaman and Hossam S. Hassanein. Core-selection algorithms in multicast routing - comparative and complexity analysis. *Computer Communications*, 29(8):998–1014, 2006.

[17] T. B. Znati and J. Molka. A simulation based analysis of naming schemes for distributed systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 42–53, Los Alamitos, CA, USA, April 1992.

[18] P. Beskow, P. Halvorsen, and C. Griwodz. Latency reduction in massively multi-player online games by partial migration of game state. *Second International Conference on Internet Technologies and Applications, Wrexham, Wales*, pages 153–163, September 2007.

[19] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: Universal topology generation from a user's perspective. Technical Report BUCS-TR-2001-003, Computer Science Department, Boston University, April 2001.

[20] Bernard M. Waxman. Dynamic Steiner tree problem. *SIAM J. Discrete Math.*, 4:364–384, 1991.

[21] B.C. Brookes. The derivation and application of the Bradford-Zipf distribution. *Journal of Documentation*, 24(4):247–265, 1968.

Paper III

Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction

Paul B. Beskow, Geir A. Erikstad, Pål Halvorsenand Carsten Griwodz

Published: Proceedings of the workshop on Network and System Support for Games (NetGames), ed. by Maha Abdallah, pp. 1-6, IEEE, 2009

Evaluation: 37 papers submitted, 10 accepted, with acceptance rate of 27.0%

Author contribution: Beskow was the the driving force behind this article and acted as the primary investigator. The master student Erikstad provided some application code.

Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction

Paul B. Beskow, Geir A. Erikstad, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory, Norway   Department of Informatics, University of Oslo, Norway
{paulbb, geirerik, paalh, griff}@ifi.uio.no

***Abstract*—Massively multi-player online games (MMOGs) have stringent latency requirements and must support large numbers of concurrent players. To handle these conflicting requirements, it is common to divide the virtual environment into virtual regions, and spawn multiple instances of isolated game areas (known as dungeons or instances) to serve multiple distinct groups of players. As MMOGs attract players from all around the world, it is plausible to disperse these regions and instances on geographically distributed servers. Core selection algorithms can then be applied to locate an optimal server for placing a region or instance, based on player latencies. Functionality for migrating objects supports this objective, with a distributed name server ensuring that references to the moved objects are efficiently maintained. As a result, we anticipate a decrease in aggregate latency for the affected players. With Ginnungagap we present the implementation and evaluation of a middleware that combines these ideas and present its feasibility.**

I. Introduction

In online games, some latency is tolerable [11] as long as it does not exceed the threshold for playability from 100ms to 1000ms depending on the type of game [13], but high pair-wise latency remains a primary obstacle for the quality of perceived game-play in a number of online games. For example, Massively Multi-Player Online Games (MMOGs) often rely on a client-server model for event distribution, where the events are collected at the server, and distributed to the interacting players. Players with high latency will inadvertently have a negative effect on the perceived quality of game play [11]. This occurs if one or more player's connections are comparatively slower, because any added delay is not isolated to the player alone. Due to a centralized server, the delay will propagate to the interacting parties and potentially result in inconsistencies, which the server must then recover from. While having minor effects on the outcome of the game, it results in a perceived deterioration to the quality of interaction [14]. As such, *low latency* for *all interacting players* is a prevalent goal.

This goal, however, contrasts other observations of online games, such that the game traffic varies strongly with time and the attractiveness of the individual game [8], [9] and that geographical dispersion of players in an online game depends heavily on the time of day [10]. Thus, in large games like Anarchy Online and EVE Online, there are always players all around the world interacting in the same game instance, although the geographical location of the large bulk of users shifts dynamically.

In [4], we proposed the initial design of a middleware supporting migration of partial game-state (where the level of game-state granularity can range from entire virtual regions, to instances, or single objects) to servers which are dynamically selected according to players locations. With GINNUNGAGAP[1], we present an implementation of this middleware. This middleware selects servers based on maximum latencies between all the players in a region and potential servers, and if the potential gain in terms of reduced latency exceeds a given threshold, the game state is migrated to this server. GINNUNGAGAP could also be applied in peer-to-peer scenarios, allowing all interacting parties to act as a server, which would mean a greater distribution of potential servers. We have not explored such a scenario in this paper, as we have focused on MMOGs that use a client-server approach, though we consider it a viable approach. Thus, we present results from experiments using this middleware running a simple game both in a lab environment and on PlanetLab. In summary, we show that the dynamic selection of servers is beneficial and that the overhead of migration and method invocation does not exceed the limits of a good perceived game play.

[1]The source code of the middleware, and logs from the PlanetLab experiments are available for download at http://simula.no/research/networks/software/

II. Background and Related Work

Migration techniques: Migration provides functionality to move entities of varying form and size (e.g., virtual machines, processes, data, code, etc.) between servers/proxies/clients in a distributed system. In [17], a virtual machine is transparently migrated with minimal impact on the user. Sprite and Mosix are *NIX based operating systems that allow for processes and their associated state to be migrated. Code migration makes it possible to defer the execution of code to an interacting party. Interpreted languages are typical for this, such as JavaScript in modern browsers and SQL in database management systems. Emerald and Chorus/COOL enable migration of objects (e.g., their code and state) during run-time execution of a program (per the object-oriented paradigm). Finally, we have eXternal Data Representation (XDR) and Boost::Serialization that provide functionality to migrate data. This is achieved by serialization, i.e., creating a binary representation of data in an application (e.g., ints, floats, chars, etc.), which is moved between instances of applications. Migration, at its various granularity levels, serves several purposes, such as dynamic load distribution, fault resilience, increasing resource locality or to facilitate system administration. We wish to achieve a combination of load distribution and increased resource locality by migrating game state to an optimal location (relative to the interacting users). Finding the correct level of granularity is important, and in online games there are two essential characteristics, 1) all code is shared, and 2) not all state related to an object needs to be migrated. Thus, we can eliminate some overhead by serializing only parts of an object. To this end, data migration accommodates both of these characteristics.

Server selection: Game server selection is an important facet of the playability for several types of online games. This is particularly true for games that are highly sensitive to latency, such as first person shooter (FPS) games. As such, the player's selection process is commonly guided by measuring dimensions that affect playability, such as latency and packet loss. This sensitivity to latency is commonly alleviated by distributing servers widely. With respect to this, Chambers et al. [7] have looked at how server selection can be

978-1-4244-5604-8/09/$26.00 ©2009 IEEE

optimized for a single client, when given a set of available servers. In a further study, Claypool [12] notes that we regularly find groups of players that wish to play together on a server, such as friends or clans (organized players). As such, he has investigated how server selection can be optimized from the perspective of a group of players. In two related studies by Armitage [1], [2], efficient ways of ranking servers in the discovery process itself are examined. These papers have in common that they consider server selection from the perspective of the player(s), and additionally assume a certain availability of servers (it is common for geographically coupled players, such as real life friends, to play against each other). For a world spanning game, however, where all users interact in the same game instance, such as an MMOG, the number of available servers often limited. In [16], Lee et al. present their heuristic for selecting a minimum number of servers satisfying given delay constraints (from the perspective of large scale interactive online games, such as MMOGs). Their aim, however, is to have well provisioned network paths in a centralized architecture. Thus, they do not consider the aspect of geographical dispersion of players. In a similar study, Brun et al [6] investigate how a server's location can influence the fairness of a game, and how selecting an appropriate server impacts this fairness. They use an objective function, which they call *critical response time* to rank the servers.

III. Implementation of Ginnungagap

The development of GINNUNGAGAP has been driven by the motivation of lowering the aggregate latency for groups of interacting players in interactive online games, such as MMOGs. To achieve this, three components were necessary, 1) a way of locating a server or proxy closer to the center of the players through core-selection, 2) a means of facilitating the migration of game-state, i.e., re-locating the players and the objects they were interacting with, and 3) allow for continuous interaction with the virtual environment, through remote method invocations (RMI), even during migration. GINNUNGAGAP supports all three components.

At its core GINNUNGAGAP runs three threads, **send**, which is responsible for transmitting middleware messages, labeled either RMI or MIG (method invocation and migration, respectively, see tables I and II); **receive**, which accepts new connections and receives transmitted messages; and **process**, which processes the messages, and activates core-selection at given intervals in the servers and proxies. In addition, all clients run a **ping** thread, which is responsible for gathering and transmitting its latency information to the proxies and servers, which is used as input to the core-selection algorithm (explained in section III-A).

All objects that support migration and RMI inherit from a base class that provides a minimal interface of methods and data that, correspondingly, must be implemented and present in all inheriting classes. The essential data in an object consists of the objects state and its identifier. The state of the object can be either `MIGRATING`, which implies that the object is in transit, and RMIs must be buffered until they can be forwarded and processed, or `NORMAL`, which implies that an RMI can be processed immediately. As an alternative to buffering messages it could be more beneficial to utilize related ideas, such as transparent migration of virtual machines [17], where one variation utilizes a two-stage approach, with an initial push-phase, where data is pushed to the receiving node, data modified during this stage is marked as dirty and is retransmitted during the stop-and-copy phase, where, in our case, the object is again made available for interaction. The unique identifier of an object follows it throughout its life-time and throughout the distributed system, and is used to uniquely locate that object at any node. A name service is necessary, however, to maintain knowledge of which node the object is currently residing at.

Message type	Object id	Object type	Attribute	Attribute
MIG	7e2...31f	CLASS_PLAYER	Odin	98

Table I
Migration message

Instrumental to the name service is an efficient way of maintaining references to the objects as they are migrated. To accomplish this we have implemented a distributed name service. A name service can be implemented in several ways. Discussed in the context of MMOGs, in [3]. The conclusion is that a distributed name service is an efficient way of handling references as there is a minimal overhead in binding an object with the name service, because each node has its own name service, to which objects are bound initially. Communication between nodes (and thus the name services) only becomes necessary when an object is migrated. Given that servers in the system are geographically distributed, binding objects locally becomes quite beneficial. Other advantages are that there is no single point of failure, so large parts of the application can continue running if a server fails. Look ups are also efficient, as we can directly query the node our name service has registered as current caretaker of the object. This access time, however, will depend on the number of times an object has been migrated (after its point of creation) and at which point in this chain the invocation is performed. We partially alleviate this situation by embedding the calling nodes' information with a message that passes through such a chain, such that the reply is sent directly back to the calling node.

A. Core selection

In [18], Knut-Helge Vik identified k-Median (see algorithm 1) as the heuristic algorithm for the graph-theoretical problem of locating an optimal proxy for a set of interacting clients, and this is the algorithm we use in the core-selection component. However, any of the node selection algorithms presented by Vik can be used instead of the k-Median algorithm in our middleware.

The k-Median core-node selection algorithm finds k core-nodes that are the k nodes with the lowest average pair-wise distances to the nodes in the member-node set. The algorithm solves the k-minimum-pairwise problem, which when given a weighted graph $G = (V, E, c)$ and an integer $0 < k < |V|$, finds a set $C \subset V$ of size k, such that the sum of the distances from the vertices's $u \in C$ to all nodes $v \in V$ is minimal. The k-Median algorithm has a time-complexity of $O(n^2)$ on any graph.

Algorithm 1 k-MEDIAN(G):

```
1: Input: An integer k > 0, a graph G = (V, E, c). Sets Z ⊂ V and X ⊂ V.
2: Output: A set C ∈ X of core-nodes.
3: map<x-id, pair-wise> mapIdPairwise
4: for each x ∈ X do
5:     pairwise = getPairwiseDistances(x, Z)
6:     mapIdPairwise.insert(x-id, pairwise)
7: end for
8: C = kLowestPairwise(mapIdPairwise)
```

B. Migration

Migration (see figure 1 for an overview) targets are found by core-selection **(1)**, though migration is only performed when latency increases above a predefined threshold. The migration is performed by the `MigrationService` **(2)**, which first sets the state of the migrating object to `MIGRATING` **(3)**. Thus, all incoming RMIs are

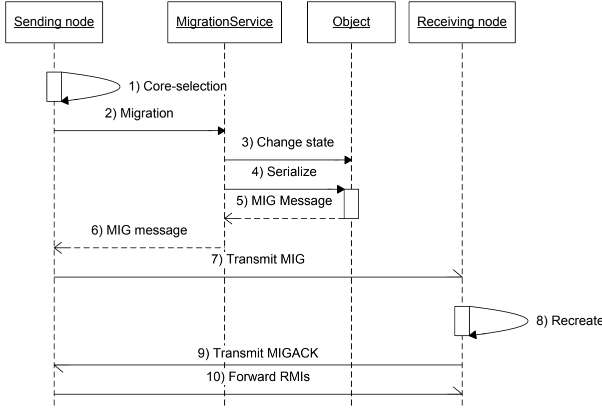


Figure 1. Sequence of a migration

Message type	Object id	Object type	Method	Parameter	Parameter
RMI	5c1...13c	CLASS_PLAYER	METH_MOVE	North	50

Table II
RMI MESSAGE

temporarily buffered locally, until they are forwarded and processed by the object at its new location. The buffered RMIs are marked as chained invocations, and as such, contain information about the node from which the call originated.

After the state update, the object is serialized **(4)**. The serialization results in the creation of a `MIG` message (see table I) in the binary XDR format **(5)**. Not all attributes of an object are necessarily serialized, but only those marked for serialization by the programmer. Once the message has been packed it is transmitted to the receiving node **(7)**.

At the receiving node, the message is processed **(8)**. The middleware reads the type of the message that it has received (`MIG`) and invokes the appropriate handler. It passes the remainder of the message to the MIG-handler, which reads the object type and requests that a new object instance of that type is created. The instance is created by calling the object constructor, which initializes the object with the attributes deserialized from the message.

Once the object has been created and initialized the MIG-handler creates a MIGACK message with the id of the object and transmits this message to the sending node **(9)**. Upon receiving this message, the sending node knows that the migration has been completed and forwards any waiting RMI-messages in its buffer **(10)**.

This same process is used to migrate groups of objects, though with a slight modification. In the case of groups, all the objects are serialized into one large message, instead of being sent in small individual ones.

C. Remote Method Invocation

An RMI is the invocation of a method on an object that doesn't reside in local memory (see figure 2 for an overview). To accomplish this goal, a number of components need to be present in the middleware. In this example, we detail the interaction of these components, starting with a player logging in to a remote server:

```
dist_ptr<Player> plyr = Startup::login(``Odin'');
```

This `login(...)` invocation is in itself an RMI, but is possible because the server and `Startup` object have a well-known location and identifier. With this invocation, the server creates a player object and returns its corresponding universally unique identifier (UUID), which is guaranteed unique for this object throughout its lifetime. The UUID is entered into the name service of the local node. Together

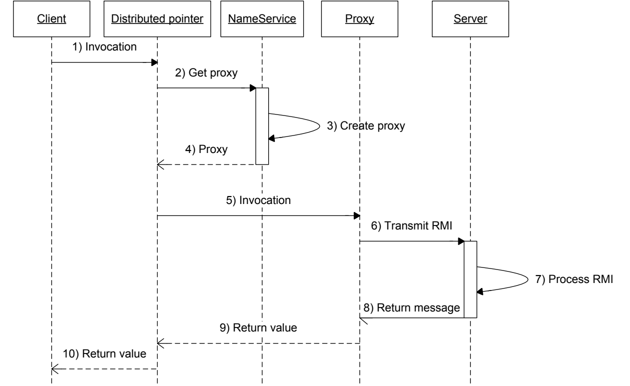


Figure 2. Sequence of a RMI

with network information about the sending node, this ensures that we have a way of contacting the remote node and identifying that instance of the object class. The UUID is also stored in the distributed pointer `plyr`. After input from the player, a request to move is issued:

```
plyr->move( Direction, Distance );
```

As `plyr` is a distributed pointer, the `move` method invocation is intercepted by the smart pointer *dist_ptr* **(1)**, which contains the UUID of the object it points to and is used to query the name service for a pointer to the object corresponding to that UUID. The name service does a look up in its records and performs one of two actions, 1) returns a pointer to the local object, or 2) returns a pointer to a proxy object if it is remote **(2)**. At the first request for a remote object a proxy is created **(3)**, and is reused for subsequent requests.

As the object we have invoked is not local, the distributed pointer redirects the call to the move-method through the proxy object with the parameters passed to the method **(5)**. Methods in an object that support RMI are implemented as virtual methods and have their own implementation in the proxy object.

At this point the proxy object takes over the processing, and creates a message of type RMI (see table II). This identifies the class type of the object, the specific object (UUID) and the function that is being invoked, and its corresponding parameters. All this information is serialized using XDR. With knowledge that the object is remote, the proxy object then transmits the message to its destination **(6)**.

At the receiving node the message is processed by the middleware **(7)**. It determines the message type (`RMI`), and invokes its corresponding handler. This RMI-handler determines if the object is local or remote, and in the latter case passes the message on (with the information of the emanating node embedded in the message). Thus, the reply, once the RMI is processed, can be sent directly to the original caller. Minimizing extraneous messages in the network. If the object is local, the corresponding skeleton method is invoked, which deserializes the parameters of the object, and invokes the method of the local object. Once the invoked method returns, the RMI has completed successfully at the remote node, and a potential return value from the function itself is serialized and returned. The middleware itself generates a reply to the calling node with information of the successful completion **(8)**. Finally, this return value is processed by the calling node **(10)**.

IV. EVALUATION

To evaluate GINNUNGAGAP, we have performed several experiments and present the vital results of our testing, with micro benchmarks of the RMI and migration functionality and live tests on PlanetLab (see [15] for further results and details).

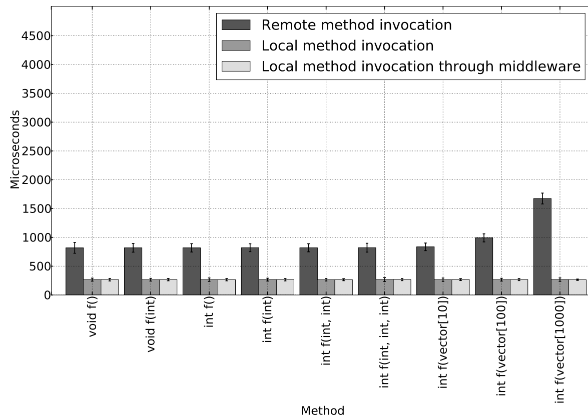


Figure 3. Average time of method invocation with stdev

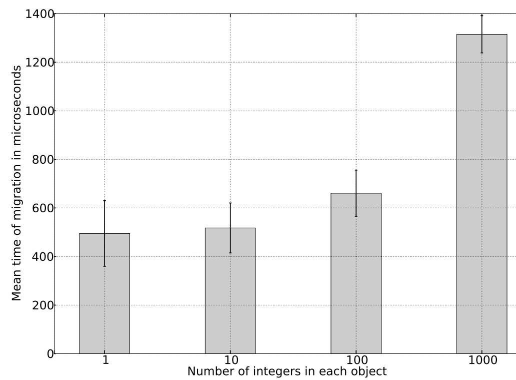


Figure 4. Ave. single object migration time with stdev

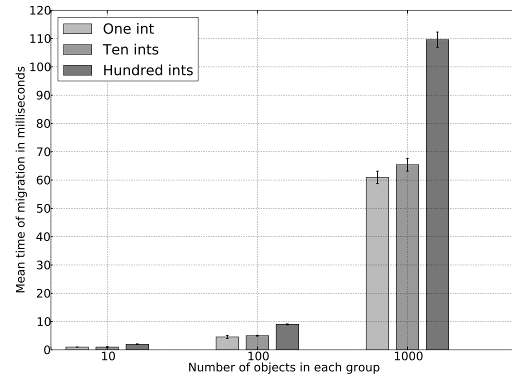


Figure 5. Ave. group migration time with stdev

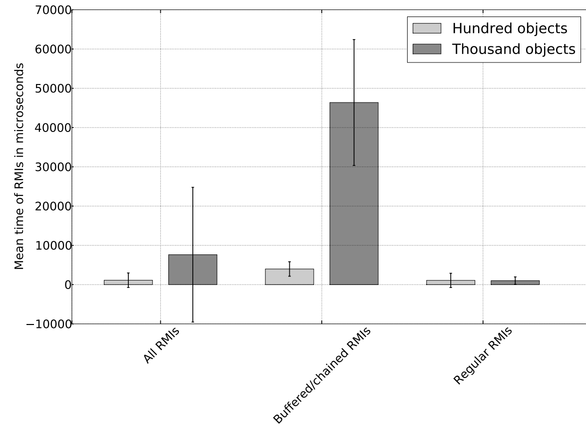


Figure 6. Average time of RMI during migration with stdev

A. Micro benchmarks

The micro benchmarks were performed using a local network. The test machines ran Ubuntu 9.04, and had identical hardware configurations, with an Intel Core 2 Duo E6750 2.66GHz CPU and 2 GB of RAM.

Remote Method Invocation: To test the RMI functionality and overhead we invoked methods with a number of different combinations of return values and parameters. The results of these tests are shown in figure 3, and each method was invoked 10000 times. As expected, there is an overhead introduced by a RMI, when compared to a local invocation, partially due to the overhead of (de)serializing parameters (not necessary for local invocations), and also the latency introduced by the network. We see a gradual increase in the RMI invocations of vectors of integers, but this is mainly due to the increased transmission time of the data. Apart from this, the mean difference in a remote and local invocation remains quite stable. We can also see that the standard deviation is relatively low. In conclusion, there is some overhead associated with an RMI compared to a local invocation, but this is to be expected. The primary obstacle is the latency introduced by the network, and the size of data being transmitted.

Migration: In the migration tests, we have performed two sets of tests, one where we migrate single objects of varying size, and one set where we vary the size of the object groups and their size.

The results of migrating a single object are shown in figure 4. As expected, the time to complete a migration of a single object gradually increases with the size of the object.

The results of migrating groups of objects are shown in figure 5. Here we see much of the same pattern as migration of single objects, with the gradual increase in completion time being mainly affected by the size of the objects and the number of groups. There is, however, more overhead introduced by the migration of groups, as the middleware has to perform additional work; it combines the objects into a single large message.

RMI during migration: We have also looked at the overhead introduced while performing RMI on an object while it is being migrated, the results of this test are presented in figure 6. We continuously performed RMI to a group of objects that were being migrated back and forth between two servers at regular intervals. As we can see, it is clear that performing an RMI during migration adds overhead. This overhead is introduced because the name service has not been updated yet, and as such RMIs are buffered until they can be forwarded. Currently, we do not forward messages as they arrive, instead we wait for a `MIGACK` before forwarding the messages. It is reasonable to assume that this overhead could be reduced by changing this to an on-the-fly forwarding of the incoming messages. The messages could then be ready to be processed as soon as the migration was completed at the receiving node, thus not having to wait for the sending node to receive the `MIGACK` message and forward the messages, effectively removing some of the overhead of the latency between the sender and receiver.

B. Planetlab tests

PlanetLab (PL) is a distributed research network consisting of nodes across the world. We ran several different tests, primarily to test the usability of the core node selection algorithm and middleware itself. Also, the PL nodes that we selected as a server and proxies were those with little or no load in the PL network.

To test the core node selection algorithm, we ran a test where clients, approximately 120 of them, were initially connected to a

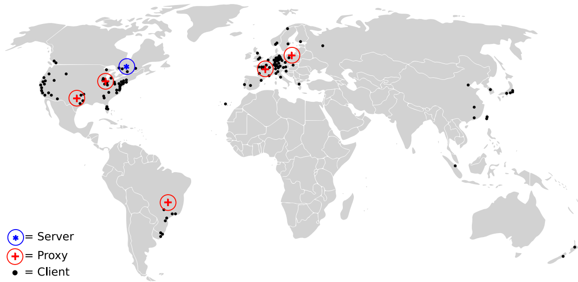


Figure 7. Geographical locations PL nodes

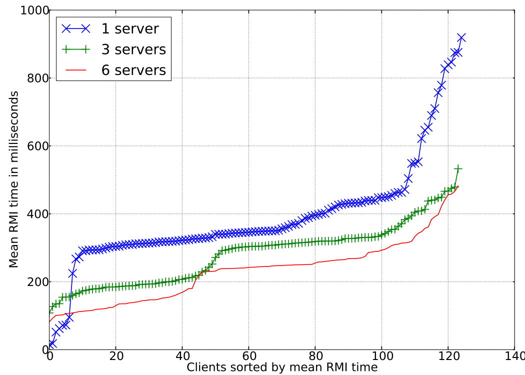


Figure 8. Aggregate mean of RMI

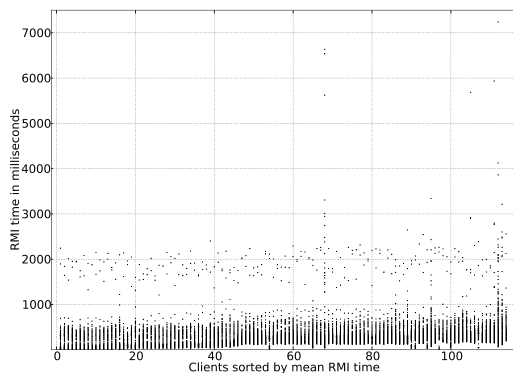


Figure 9. RMI time per client with varying numbers of proxies

main server. As more proxies were gradually added, the core node selection algorithm was activated (see figure 7 for an overview of the PL nodes included in the experiment). The results of this test are summarized in figure 8 and 9. As we can see in figure 8, adding additional proxies made it possible to reduce the aggregate mean latency of the interacting clients. Moving up to four proxies further decreased the aggregate mean latency, but the gain was not as great. It is to be expected that the reduction does not necessarily improve greatly with a large number of added proxies, as there is always variance in the latencies of the connected clients, but it is clear that the gaming experience can be improved by the introduction of a small number of proxies.

In figure 9, we see the completion time of RMIs issued, as proxies are being added. We can see that most of the invocations are well below the 1000ms mark, but that there are at least a couple of invocations that take longer time. These are caused by migrations, as we noted during our micro benchmarks of RMI during migration, as seen in figure 6.

In figure 10, we see the results of a test simulating a small

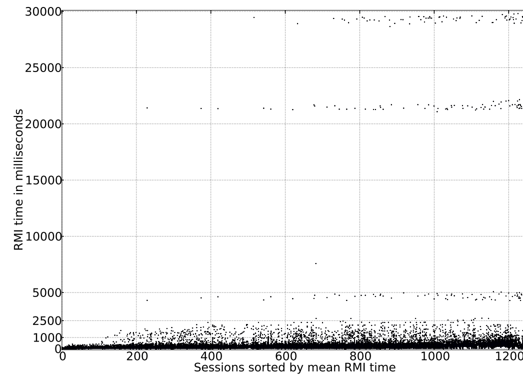


Figure 10. Mean RMI time of sessions

region in a virtual environment with players entering and exiting the region continuously. The setup consisted of 1 server, 5 proxies and approximately 120 clients. The session time of the clients follows an exponential distribution with a scale of 60 seconds. The core-selection algorithm was activated every 30 seconds, with a migration occurring only if the minimum gain of each client was 2ms. This resulted in a total of 23 migrations, where all, except the first (from the server to a proxy) were between two of the proxies in the setup. The mean migration time was 813ms, excluding one exception, which was on 28868ms, and we can see the resulting spikes in the RMI times of the sessions in figure 10. PL uses virtualization to schedule slices for run-time on the PL nodes. We believe this spike is a result of such virtualization, where the slice running our application has been unscheduled. We did not see such spikes in our earlier PL tests, but these ran for shorter periods of time. This is one of the problems with running latency sensitive experiments in PL. A ping between the two proxies involved did not reveal any problems latency wise, but is not affected by such virtualization. We ran the test several times with different configurations, but experienced similar spikes each time. We did not experience these in our micro benchmarks of the migration functionality either. Apart from these spikes, most of the invocations during this test were well within the 1000ms mark. As such, it should be possible to perform migrations on regions with heavy traffic quite frequently, without adverse side-effects.

V. Discussion

Due to the distributed name service resolving references to remote objects, migration can become a time consuming task. An object that is frequently migrated will create trails of object references at the nodes it visits. We partially handle this scenario by embedding information about the emanating node in a message that is forward, so the reply can be sent directly to the calling node.

Another concern is related to the side-effects of migrating game-state. If migration is not performed transparently, we might adversely affect interacting players. In [17], Nelson et al demonstrate how an active application is moved from one virtual machine to another, with minimal perceived impact to the user's interaction with the application.

Partial failures occur when a node in a distributed system becomes unavailable, effectively rendering the objects managed by it inaccessible. The current framework does not accommodate for partial failures, but there are ways to minimize the repercussions of these incidents. One possibility is to distribute multiple copies of an object in the system, and utilize a peer-to-peer based look-up system such as Chord or Tapestry, to locate the object. Though this introduces the overhead of having to keep multiple objects up-to-date.

Core selection and migration require resources in the form of processing power and network messages when activated, as such, policies should govern their activation. Core-selection can be performed on the basis of churn (in the form of players joining/leaving), time of day, player arrival rate, history based (a preemptive approach, where known situations, such as battles, trigger the activation) and server-load. Migration can be performed on the basis of threshold of aggregate latency, number of players, packet loss, and server load.

Core selection is dependent on full knowledge of the network, but obtaining latency measurements through active probing is not a scalable solution. Latency estimation techniques provide an alternative, and in [5], [19], we examined the feasibility of using estimated latencies for situations when real-time measurements are unable to scale to the number of interacting players. We have not applied these techniques in this paper, as we were able to gather live measurements during our tests.

The proposed framework is tested in a client-server setting, but in general, the framework should also be useful in a peer-to-peer scenario trying to dynamically select a server among the peers. Thus, as the peers join and leave the game, the selection of peers controlling the game could be performed using our middleware.

VI. Conclusion

High latency may be devastating for the game play experience. An important factor for world-spanning games, such as MMOGs, lies in the diversity of its user base where there will, at any point in time, be a number of players connected from different physical locations. The large challenge is that the group of players in the game or in the game region is dynamic, and using a statically selected server may therefore give good results in one moment but poor in the next.

In [4], we proposed a middleware providing dynamic server selection and migration of game state to the new server. Here, we have described the implementation of this middleware and presented experimental results showing the performance gains from in-house lab experiments and running a simple game on PlanetLab.

References

[1] G. Armitage. Client-Side Adaptive Search Optimisation for Online Game Server Discovery. *Lecture notes in computer science*, 4982:494, 2008.

[2] G. Armitage. Optimising online fps game server discovery through clustering servers by origin autonomous system. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, May 2008.

[3] P. Beskow. Migration of objects in a middleware for distributed real-time interatctive applications. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2007.

[4] P. Beskow, K.-H. Vik, C. Griwodz, and P. Halvorsen. Latency reduction by dynamic core selection and partial migration of game state. *Proceedings of NetGames'08, Worcester, MA, USA*, oct 2008.

[5] P. Beskow, K.-H. Vik, P. Halvorsen, and C. Griwodz. The partial migration of game state and dynamic server selection to reduce latency. *Springer's Multimedia Tools and Applications Specia l Issue on Massively Multiuser Online Gaming Systems and Applications*, 2009.

[6] J. Brun, F. Safaei, and P. Boustead. Server topology considerations in online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 26, New York, NY, USA, 2006. ACM.

[7] C. Chambers, W. chang Feng, W. chi Feng, and D. Saha. A geographic redirection service for on-line games. *In Proceedings of the eleventh ACM international conference on Multimedia, Berkeley, CA, USA*, pages 227–230, November 2003.

[8] C. Chambers, W. chang Feng, S. Sahu, and D. Saha. Measurement-based characterization of a collection of on-line games. *In the Proceedings of the 5th ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA*, pages 1–14, October 2005.

[9] W. chang Feng, F. Chang, W. chi Feng, and J. Walpole. Provisioning on-line games: a traffic analysis of a busy Counter-strike server. *In the Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement, Marseille, France*, pages 151–156, November 2002.

[10] W. chang Feng and W. chi Feng. On the geographic distribution of on-line game servers and players. *In the Proceedings of NetGames'03, Redwood City, California, USA*, pages 173–179, May 2003.

[11] M. Claypool. The effect of latency on user performance in real-time strategy games. *Elsevier Computer Networks*, 49(1):52–70, Sept. 2005.

[12] M. Claypool. Network characteristics for server selection in online games. *In Proceedings of the fifteenth Annual Multimedia Computing and Networking (MMCN'08), San Jose, CA, USA*, 6818:681808, January 2008.

[13] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, Nov. 2005.

[14] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. *In the Proceedings of NetGames'05, Hawthorne, NY, USA*, pages 1–7, October 2005.

[15] G. A. Erikstad. Latency reduction in distributed interactive applications by core node selection and migration. Master's thesis, Department of Informatics, University of Oslo, Norway, Aug. 2009.

[16] K. Lee, B. Ko, and S. Calo. Adaptive server selection for large scale interactive online games. *Computer Networks*, 49(1):84–102, 2005.

[17] M. Nelson, B. Lim, and G. Hutchins. Fast transparent migration for virtual machines. *Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference table of contents*, pages 25–25, 2005.

[18] K.-H. Vik. *Group Communication Techniques in Overlay Networks (submitted)*. PhD thesis, Department of Informatics, University of Oslo, Norway, Dec. 2008.

[19] K.-H. Vik, C. Griwodz, and P. Halvorsen. On the influence of latency estimation on dynamic group communication using overlays. In *Multimedia Computing and Networking (MMCN), San Jose, CA, USA*, January 2009.

Paper IV

The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency

Paul B. Beskow, Knut-Helge Vik, Pål Halvorsenand Carsten Griwodz

Published: Springer's Multimedia Tools and Applications, An International Journal (MTAP) 45(1-3):83 - 107, 2009

Evaluation: This journal paper was submitted by invitation to Springer's Multimedia Tools and Applications (MTAP) Special Issue on Massively Multiuser Online Gaming Systems and Applications.

Author contribution: Beskow was the primary investigator and contributor to this journal paper. Vik provided input on the core-selection side. All application code was developed by Beskow.

Noname manuscript No.
(will be inserted by the editor)

The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency

Paul B. Beskow · Knut-Helge Vik · Pål Halvorsen · Carsten Griwodz

Received: date / Accepted: date

Abstract Massively multi-player online games (MMOGs) have stringent latency requirements and must support large numbers of concurrent players. To handle these conflicting requirements, it is common to divide the virtual environment into virtual regions. As MMOGs are world-spanning games, it is plausible to disperse these regions on geographically distributed servers. Core selection can then be applied to locate an optimal server for placing a region, based on player latencies. Functionality for migrating objects supports this objective, with a distributed name server ensuring that references to the moved objects are maintained. As a result we anticipate a decrease in the aggregate latency for the affected players. The core selection relies on a set of servers and measurements of the interacting players latencies. Measuring these latencies by actively probing the network is not scalable for a large number of players. We therefore explore the use of latency estimation techniques to gather this information.

Keywords Massively multi-player online games · latency · estimation · server architecture

1 Introduction

"Lagger!" is a likely expression to hear uttered in a real-time interactive online game. This term addresses players with excessive latency, which in the gaming community is colloquially referred to as *lag* (no positive connotation is implied by this term). A player with

Paul B. Beskow (paulbb@ifi.uio.no), Knut-Helge Vik (knuthelv@ifi.uio.no)
Pål Halvorsen (paalh@ifi.uio.no), Carsten Griwodz (griff@ifi.uio.no)

Simula Research Laboratory
Martin Linges v 17
N-1364 Snarøya
Telephone: +47 67828200
Fax: +47 67828201

Department of Informatics, University of Oslo
Gaustadalléen 23
N-0373 Oslo
Telephone: +47 22852410
Fax: +47 22852401

high latency will inadvertently have a negative effect on the perceived quality of the game play [4, 15]. This occurs, as most online games are based on a client-server model, where events are collected at the server, and distributed to the interacting players. By its nature, if one player's connection is comparatively slower, any added delay is not isolated to the player alone. It will propagate to the interacting parties and potentially result in inconsistencies, which the server must then recover from. While having minor effects on the outcome of the game, it results in a perceived deterioration to the quality of interaction [20]. As such, *low latency* for *all players* is a prevalent goal. In this paper, we extend our body of work on migration and server selection [7] and expand the latter to include an exploration of latency estimation techniques to obtain the required network information in a scalable way. Thus, we use migration techniques to move the processing to a more appropriately placed node found using a core selection algorithm based on obtained network information.

The latency requirements of games vary greatly, ranging from 100 to 1000 ms [17]. One factor affecting the latency seen by the players is their physical distance to the server. As such, to achieve a satisfactory quality of interaction, the player may need to be located within a reasonable proximity to the game server, or that we have a dynamic selection of servers; to be in the proximity of most of the users.

As an example, consider world-spanning MMOGs, which are persistent online worlds that allow thousands of players to interact concurrently in a virtual environment. To support this many concurrently interacting players, the virtual environment is commonly split into virtual regions. This makes it possible to distribute the regions across a number of (possibly geographically distributed) servers. As the regions are logically decomposed, each server is responsible for handling some regions and the players interacting in each region. Since a player's proximity to the server impacts the latency, and in turn, latency is an integral factor for the playability of an online game, this raises an interesting question: *How can we optimize the placement of a virtual region given the interacting players and a set of globally distributed servers?*

Core selection provides a solution to this server selection problem. Given a set of players and servers (and proxies), it finds an appropriate server (or proxy) for placing a virtual region. It measures the latency of each player to each server and locates the server that provides the minimum diameter (lowest of the highest pair-wise latencies) in order to take into account that a game event sent from one player must reach all other players within the latency constraints.

Core selection depends on latency measurements from the servers to the players. These latencies may be obtained by actively probing and monitoring the network, but this is not scalable due to the potentially large number of players and servers/proxies, i.e, n^2 measurements in the worst case. As such, we investigate the use of latency estimation techniques for use with core selection, focusing on Vivaldi [19] and Netvigator [41]. Estimation techniques measure a sub-set of the links, and then estimate the remaining links based on these measurements. We consider the impact of estimation on the quality of the core selection process, because although these techniques are scalable, a penalty arises from their estimation accuracy. Results show that Netvigator yields accurate latency estimates, while Vivaldi is more inaccurate, but still usable. Netvigator is harder to set up than Vivaldi, but they are both likely candidates for use in distributed interactive applications.

Once a server has been selected, we use our migration functionality to move the active region to its new location. To maintain an optimized set of references to the migrated game state, we use a distributed name service. On the background of existing work in distributed systems, we believe that the combination of *core selection, latency estimation, a distributed*

name server and migration, a viable solution for creating a globally distributed game, which is capable of *lowering the overall latency of the interacting players.*

The rest of the paper is organized as follows: in section 2, we look at the basis for our assumptions, and take a closer look at some related work. In section 3, we look at the migration functionality and how it is supported by the distributed name service. In section 4, the core selection process is described in detail, and how it can be applied to the scenario we have described. In section 5, we describe different classes of latency estimation techniques. In section 6, we evaluate our migration functionality and the core selection process (using both measured and estimated input). In section 7, we discuss major aspects of this work. Finally, we summarize our findings in section 8.

2 Background and Related Work

2.1 Game Characteristics

The body of work that analyzes game traffic has grown considerably in the recent past. The main conclusions in our scenario are that 1) game traffic varies strongly with time and the attractiveness of the individual game [12, 13], 2) some latency is tolerable [15] as long as it does not exceed the threshold for playability from 100ms to 1000ms depending on the type of game [17] and 3) geographical dispersion of players in an online game depends heavily on the time of day [25]. In addition to these works, we have analyzed packet traces (see [29]) from Funcom's popular role-playing MMOG *Anarchy Online* [26]. Statistics from the traces reveal that there lies a potential latency improvement in using our migration middleware. For example, in one of the game regions, with measurements recorded within a time span of about one hour, we found approximately 175 distinct connections. These are sorted according to their measured round-trip times (RTTs) in figure 1. With the knowledge that the servers are located in the US, the observed minimum latencies in the figures indicate that there are players concurrently located in the US, Europe and Asia. The number of players in different areas of the world also typically vary according to the time of day, and finding an appropriate location for the server might be of vital importance in order to meet the latency requirements. In figure 1 we can see that the majority of users are in the second range (20-140), as such, the average latency could be reduced by moving the analyzed game region to a server in Europe.

2.2 Migration techniques

Migration provides the functionality to move entities between servers/proxies/clients in a distributed system. These entities can take different forms and be of greatly varying granularity. In [14] Clark et al. migrate a live virtual machine from one node in a cluster to another, with as little as 60 ms of downtime. In a similar study, Nelson et al. [36] describe how a virtual machine can be transparently migrated with minimal impact on the user. At a lower granularity level we find DEMOS/MP [39], Sprite [33] and Mosix [3], which are all *NIX based operating systems that allow for processes to be migrated. They are required to capture all of the data and state associated with the process (the program data, its stack and registers: program counter, stack pointer, and so on) to facilitate the migration. JavaScript and SQL [23] are examples of systems that make code migration possible. JavaScript is client-side executed code, while SQL is primarily executed by the database management system

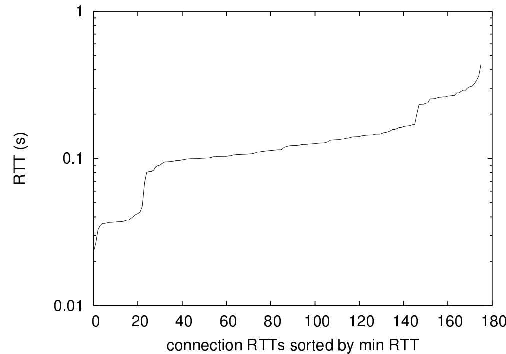


Fig. 1 Anarchy Online: connection RTTs sorted by min RTT

at the server hosting the database system. Finally, Emerald [30] and Chorus/COOL [8] are examples of systems that make it possible to migrate objects during run-time execution of a program (per the object-oriented paradigm). Migration can be used to serve several purposes, such as dynamic load distribution, fault resilience, increasing resource locality or to facilitate system administration. In our case, we wish to achieve a combination of load distribution and increased resource locality by migrating game state to an optimal location (relative to the interacting users).

2.3 Server selection

Game server selection is an important facet of the playability for several types of online games. This is particularly true for games that are highly sensitive to latency, such as first person shooter (FPS) games. As such, the player's selection process is commonly guided by measuring dimensions that affect playability, such as latency and packet loss. This sensitivity to latency is commonly alleviated by having a high distribution, and availability, of servers. With respect to this, Chambers et al. [11] have looked at how server selection can be optimized for a single client, when given a set of available servers. In a further study, Claypool [16] notes that we regularly find groups of players that wish to play together on a server, such as friends or clans (organized players). As such, he has investigated how server selection can be optimized from the perspective of a group of players. In two related studies by Armitage [1,2], efficient ways of ranking servers in the discovery process itself are examined. These papers have in common that they consider server selection from the perspective of the player(s), and additionally assume a certain availability of servers (it is common for geographically coupled players, such as real life friends, to play against each other). For a world spanning game, however, where all users interact in the same game instance, such as an MMOG, there is often a limited number of servers to select from. In [34], Lee et al. present their heuristic for selecting a minimum number of servers satisfying given delay constraints (from the perspective of large scale interactive online games, such as MMOGs). Their aim, however, is to have well provisioned network paths in a centralized architecture. Thus, they do not consider the aspect of geographical dispersion of players. In a similar study, Brun et al [10] investigate how a server's location can influence the fairness of a game, and how selecting an appropriate server impacts this fairness. They use an objective function, which they call *critical response time* to rank the servers.

2.4 Network estimation

It is straightforward to obtain a link's RTT in the Internet using measurement tools like ping and traceroute (or mechanisms like packet pair and packet train [21]). The drawback with ping is that it does not return any measurements if the target host is unreachable. Traceroute, on the other hand, returns latency measurements for each (answering) hop on the route. This may be valuable for some latency estimation techniques if they control network routers, and if the end-to-end reachability is limited. Furthermore, in the scenario of large-scale distributed interactive applications, it is currently not scalable to use measurement tools like ping or traceroute to actively monitor networks for their link latencies. Instead, latency estimation techniques that reduce the probing overhead should be applied.

Achieving full up-to-date knowledge of the network requires live latency monitoring and is not scalable for a large number of players and servers/proxies. In the best case $M \times N$ (for M servers/proxies, and N players) measurements are required, in the worst case N^2 (in the case where all nodes can be used as servers, i.e., a peer-to-peer setting) are required. This scalability problem is addressed by techniques that estimate link latencies, but the trade-off is their accuracy. In general, a latency estimation technique probes a (low) number of links to sample their link latencies, and then attempts to estimate the remaining links based on these probes. There are a multitude of latency estimation techniques. Many of the latency estimation techniques are likely to be usable in a distributed interactive application setting. However, the main comparative metric is whether or not the estimations are accurate enough.

The estimation techniques may be classified into one of three classes [24].

Landmarks-based latency estimation technique assign each node a point in a metric space, and aim to predict the latency between any two nodes. They use landmark nodes, a set of nodes that are used by others as measurement references for their relative position in the network. Examples of landmarks-based techniques are Netvigator [41], NetForecast [24], Global Network Positioning (GNP) [37] and Practical Internet Coordinates [18].

Multidimensional-scaling based latency estimation techniques use statistical techniques for exploring similarities and dissimilarities in data. For example, a matrix of item-item similarities is used to assign a location for each item in a low-dimensional space [18]. Vivaldi [19] is such a technique.

Finally, *distributed network latency database* techniques use active measurements to build a knowledge base about the underlying network. These approaches have been designed to efficiently answer queries of the form: Who is the closest neighbor to node A in the network? Since these schemes are based on direct measurements they have better accuracy. They also inject more traffic into the network compared to the landmark-based and multidimensional-scaling based techniques. Meridian [45] is a technique that uses a distributed network latency database.

Distributed network latency database techniques are not desirable for our target area, because they are not designed to retrieve all-to-all link latencies. Instead, however, we focus on landmarks-based and multidimensional-scaling based latency estimation techniques. They are desirable for leader election scenarios, and discovering the closest neighbor for a node.

2.5 Summary

Our analysis of the Anarchy Online game traffic shows that players connect from all around the world (see figure 1). An approach to reducing the latency, both due to RTT and loss, is to

dynamically find the center of the group of players and migrate the game objects to a server whose location is closer to the majority of the users. We can accomplish this reduction through core selection, which helps us determine which node to migrate the players to. An issue with core selection is that it depends on latency measurements to run correctly. Actively measuring these latencies with ping or traceroute is costly, and depending on the number of players, not scalable either. As such, latency estimation techniques offer a viable alternative to gather such information.

3 Moving worlds with migration

Core selection is applied to determine whether the current server hosting a virtual region (based on its player population) is optimal. In the case where it is able to locate a more appropriate server based on the latency to the active users, the MMOG moves the game state of that region to its new location. To accommodate this process, we have developed a middleware that is capable of performing such migration of game state, which consists of a number of interacting objects (following the object-oriented paradigm). Before migrating an object, we must know that all references to that object are maintained. To accomplish this, we use a name service. This service is responsible for keeping an up-to-date index for the location of objects in the distributed system, and redirect method invocations accordingly. As such, the reduction in aggregate latency is accomplished by decreasing the average response time of remote method invocations (RMI). The response time is decreased because we move objects closer to the majority of the players. To accommodate the development cycle we have also written a tool that generates code to ease integration of the application with the middleware.

3.1 Name service

A name service maintains references to objects in the distributed system. This task can be accomplished in several ways, and Znati et al [47] analyzed three approaches to implementing a name service; in the form of centralized, hybrid and distributed versions, which we have discussed thoroughly (in the context of MMOGs) in [5]. The conclusion is that the centralized model achieves acceptable performance only as long as the ratio of remote to local requests is kept reasonable. The performance of the hybrid model depends highly on the efficiency of the cache design, and with all other network conditions equal the relative response times of the distributed architecture were smaller. A distributed name service best suits our needs, primarily because efficiency is more important then consistency in this scenario. In addition, the following characteristics also have an impact on this decision:

1. There are thousands of concurrently interacting players.
2. The virtual environment is divided into virtual regions.
3. Players are dispersed physically as well as virtually.
4. The servers in the system are geographically distributed.
5. Code is shared so only data is migrated.
6. There occurs frequent object creation and destruction.
7. Efficiency is more important then consistency.

A distributed name service is more efficient then a centralized or hybrid approach because there is no overhead in binding an object with the name service. This, because each

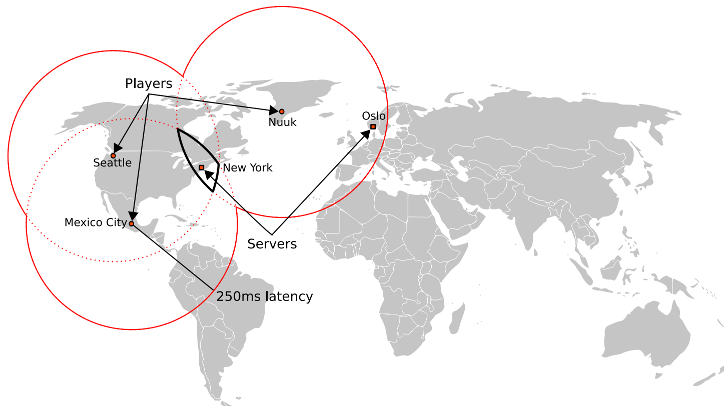


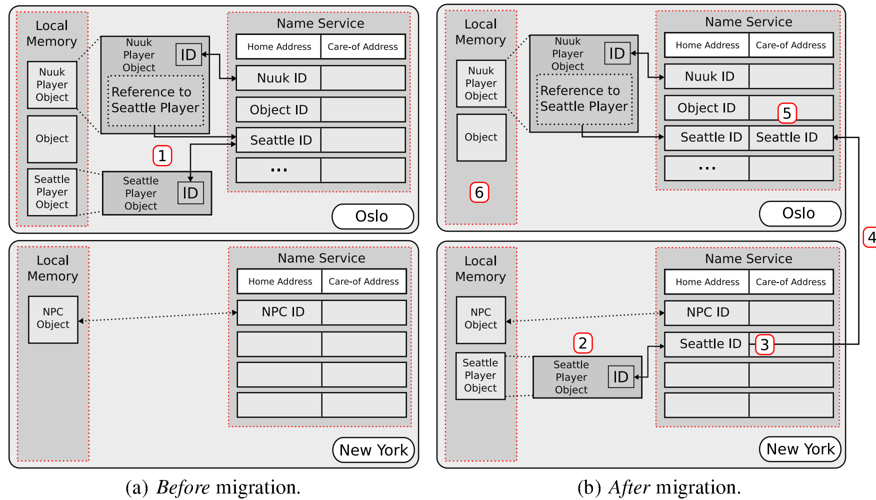
Fig. 2 Latency as physical distance

node has its own name service, to which objects are bound initially. Communication between nodes (and thus the name services) only becomes necessary when an object is migrated. Given that the servers in the system are geographically distributed, binding objects locally becomes quite beneficial. Other advantages are that there is no single point of failure, which implies that large parts of the application can continue running if a server were to fail. Looking up objects is also efficient, as we can directly query the node our name service has registered as the current care taker of the object. Though it is worth noting that this access time will depend on the number of times an object has been migrated (after its point of creation) and at which point in this chain the invocation is performed. For further details and a thorough discussion about the implementation of the name service, distributed references and migration see [6]. In the following section, we will solidify our understanding of the described mechanisms by looking at example of these concepts in use.

3.2 In action

Consider a system consisting of two servers, as seen in figure 2. The majority of the interacting players are European, with a couple of players connected from *Nuuk* and *Seattle*. Currently, all of the virtual regions are hosted in *Oslo*. Looking at figure 3(a), we see that at least two players have been bound to their local name service and received an identifier (we do not show the other players in this example). Additionally, the *Nuuk* player references the *Seattle* player. This reference is not to the player object in local memory, as we might expect, but to the identifier (created earlier) in the name service. After some time the population density shifts towards an American dominance, triggering a core selection for each virtual region (we describe this in further detail in section 4). For the region with our two players, the server in *New York* is deemed a better fit. A migration is triggered, with our two players migrated to the server in *New York*. The steps outlined in figure 3(a) (before) and 3(b) (during/after) guide us through this process.

First (1) we serialize the player object, which means a binary representation of the object is created. The data representing the serialized object is then transmitted to the receiving node, in this case the server in New York. The *second* step (2) consists of deserializing the object at the receiving side. This recreation is accomplished by initializing an object with



(a) *Before* migration.

(b) *After* migration.

Fig. 3 Server configuration

the binary stream. Once the object has been recreated, it is bound to the local name service, which is the *third* step (3). At this point in time no reference between the original node and the new node exists. As such, the original node will have no way of forwarding requests. To resolve this problem, the new identifier assigned to the player object by the server in New York is returned to the original server in Oslo, which concludes the *fourth* step (4). After Oslo has updated its name service by associating the received identifier with the player object, which happens in step *five* (5), we can without worry remove the object from local memory; as is done in step *six* (6).

3.3 Code generation

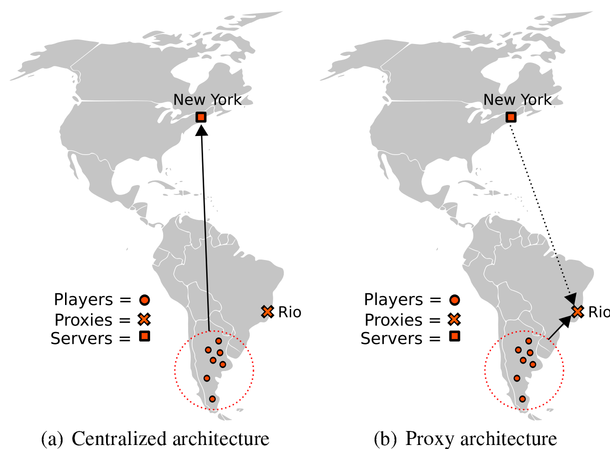
Integrating code with a middleware, such as the one we have developed for migrating objects, can be tedious and error-prone work. To accommodate the development cycle we have written a tool for automatically generating skeletons that integrate with the middleware. The code generation tool is implemented as a Python script, which takes annotated C++ header files as input. To parse the C++ header files, we make use of the GCC-XML [32] parser, which is a tool that extends the open source GCC compiler, using its internal representation to produce XML output. Based on information obtained by processing the XML-file, we generate the required code to integrate the user-defined class seamlessly with the middleware. As mentioned, the skeleton generator expects a C++ class declaration as its input. In addition to normal C++ class syntax, GCC-XML allows for defining additional attributes. We make use of this ability to extend the C++ syntax with our own keywords. When an object is migrated, one does not necessarily want all the data to be serialized. We therefore provide a special keyword (*_serialize*) to specify the data to be serialized. Other suitable keywords will be introduced to support remote method invocation, mark the classes that can be migrated and so forth.

4 Core selection

Today's typical client/server model makes it easy to manage the global game state, but it has drawbacks. The server is a potential bottleneck, both in terms of computing and bandwidth capacity, and the latency depends heavily on the physical distance from each individual player to the server. In figure 4(a), we illustrate an example where a centralized server stores the game state and thus cannot take into account the physical location of the players.

Proxy technology is a distributed option. In this model, we have an infrastructure with a centralized server and a set of distributed proxy servers. The proxies are used to increase the physical distribution of servers, where we aim to achieve much the same as content distribution networks (CDNs), i.e., have a distribution point in close proximity to the players for faster distribution of data. In this scenario, an efficient way to reduce latency can be to migrate game state to an appropriate server, close to the center of a given group of players. Consider an MMOG with geographically distributed servers (see figure 2). At any time, a server can be hosting none, some or all of the virtual regions making up a virtual environment of a game instance. Furthermore, looking at figure 2, we can see how latency (reasonably) compares to a measurement of physical distance. In case of a 250 ms delay requirement from the server (e.g, according to the 500 ms pairwise latency requirement in RTS [15]), the bordered intersection indicates an optimal area for the server to be located. In figure 4(b), the central server has migrated the game state to a proxy that is closer to the group of players.

Proxy technology allows a trade-off between client/server and peer-to-peer advantages and disadvantages. A pure peer-to-peer architecture then implies that the game state is distributed among the players, with no central server (necessarily) involved. This makes it very hard to administrate the game state such that it is consistent, and additionally there is no working business model for such a scheme.



(a) Centralized architecture (b) Proxy architecture

Fig. 4 Architecture

4.1 Core selection algorithms

It is desirable to determine if the server currently hosting a region is the optimal choice. As such, we wish to make sure whether there is a server in the system that would be able to provide these players with better overall performance in terms of network delay. To accomplish this, we can use a core selection algorithm, which determines which server provides the optimal placement for that region and its players.

The core selection algorithms are devised from a graph theory perspective. Upon core selection, the core nodes may be used to administrate players that join and leave groups. Such groups can be defined and updated dynamically in an MMOG, for example, based on some area-of-interest management (like existing in the same virtual region). Typically, the core node of a group is contacted for each membership change, such that it always has the latest view. When a limited set of nodes handles the membership management, it simplifies membership updates; applications with highly dynamic groups, such as MMOGs, require fast and simple group management.

Core-based protocols work on the assumption that one or more core nodes are selected as group management and forwarding nodes. Therefore, the cores need to be selected using a core selection algorithm. Several core selection algorithms have been proposed, and a comprehensive study is given by Karaman and Hassanein [31]. An overall goal is to select cores on the basis of certain node properties, such as, bandwidth and computational power. We wish to base this decision primarily on latency. The cores that are selected depend on the group size and location, as well as the capacities in the available core nodes. In this paper, our focus is on electing a single core node for each defined group. The core may, for example, be a server or a proxy administrated by the game provider.

The core selection algorithms presented below search among a predefined set of servers and proxies to find one optimal core, which is the *graph median*. The graph median is the node for which the sum of lengths of shortest paths to all other vertices's is the smallest. The algorithms are [31]:

– *Topology Center:* Find a central entity (server) that is closest to the topological center of the *global graph*.
– *Group Center:* Find a proxy that is closest to the group center of the *group graph*.

Topology center is given as input a set of available servers that are located around the world. The algorithm searches for the server for which the sum of latencies of shortest paths to all players in its member network is the smallest, which is comparable to finding an optimal server for an instance of the game world for all connected players. The group center algorithm similarly searches among a set of available proxies located around the world. It is given a group of players as input, and based on this the algorithm selects the core proxy to be the proxy for which the sum of latencies of shortest paths to all the players in the group is smallest, i.e., locating an optimal placement for a subset of the players (interacting in a region of the game instance, for example). These simple algorithms form powerful techniques in the search for suitable hosts to migrate game state to. We have tested several algorithms [43] and present the two most prominent, in the following. Both k-Median and k-Center find optimal solutions to two different graph theoretical problems.

4.1.1 k-Median core-node selection algorithm

The k-Median core-node selection algorithm finds k core-nodes that are the k nodes with the lowest average pair-wise distances to the nodes in the member-node set. The algo-

rithm solves the k-minimum-pairwise problem, which when given a weighted graph $G = (V, E, c)$, and an integer $0 < k < |V|$, finds a set $D \subset V$ of size k, such that the sum of the distances from the vertices $u \in D$ to all nodes $v \in V$ is the smallest. The k-Median algorithm has a time-complexity of $O(n^2)$ on any graph.

Algorithm 1 k-MEDIAN(G):

1: Input: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
2: Output: A set $C \in X$ of core-nodes.
3: map<id, pair-wise> mapIdPairwise
4: **for** each $x \in X$ **do**
5: T = ShortestPathTree(x, G)
6: v = maxEccentricityNode(T, Z)
7: pairwise = getPairwiseDistances(x, T)
8: mapIdPairwise.insert(x, pairwise)
9: **end for**
10: C = kLowestPairwise(mapIdPairwise)

4.1.2 k-Center core-node selection algorithm

The k-Center core-node selection algorithm finds k core-nodes that are the k nodes with the lowest maximum distance (eccentricity) to a node in the member-node set Z. It solves the k-minimum-eccentricity problem, which when given a weighted graph $G = (V, E, c)$, and an integer $0 < k < |V|$. Find a set $D \subset V$ of size k, such that the sum of the eccentricities yielded by the vertices $v \in D$ is the smallest. The k-Center algorithm has a time-complexity of $O(n^2)$, when run on a complete graph.

Algorithm 2 k-CENTER(G):

1: Input: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
2: Output: A set $C \in X$ of core-nodes.
3: map<id, eccentricity> mapIdEcc
4: **for** each $x \in X$ **do**
5: T = ShortestPathTree(x, G)
6: v = maxEccentricityNode(T, Z)
7: ecc = getEccentricity(v, T, Z)
8: mapIdEcc.insert(x, ecc)
9: **end for**
10: C = kLowestEccentricities(mapIdEcc)

5 Latency estimation

One important issue related to centralized core-node selection algorithms is that they need all the required network information to be available at the executing node. As monitoring the entire network is too expencive and does not scale, latency estimation techniques are important to enable centralized core-node selection algorithms in large-scale applications, such as MMOGs. When latency estimates are available, the issue then becomes how these latency estimates affect the performance of the core-node selection algorithms, which we evaluate in

Technique	*Measurement Overhead*	*Requires*	*Churn recovery*	Infrastructure dependability
Vivaldi	-	Inter-nodes traffic	yes	no
Netvigator	$O(L * N)$	Traceroute	no	yes

Table 1 Properties of the latency estimation techniques.

section 6.3.2. We have discussed different classes of latency estimation techniques (in section 2.4), where we concluded that landmarks- and multidimensional-scaling based latency estimation techniques are the most appropriate for the problem domain. Netvigator [41] and Vivaldi [19] are two highly valued techniques in their respective latency estimation technique classes [24]. Table 1 provides a small comparison.

Vivaldi is a multidimensional scaling technique and is based on spring embedding, which models network nodes as masses connected by springs (links) and then relaxes the spring length (energy) in an iterative manner to reach the minimum energy state for the system. All nodes joining the system are placed at the origin, and start sharing Vivaldi information with selected nodes piggybacked on application level data. The Vivaldi information includes its coordinates, confidence estimations and the measured latency. If a global graph is desired, each node can report its Vivaldi information to a repository that does some calculations and inserts the node in a two-dimensional plane where the Euclidian distance equals the estimated latencies. Vivaldi has the advantage that it recovers from node churn (nodes joining and leaving), and does not depend on any infrastructure.

Netvigator, often considered the most accurate [41], on the other hand, needs landmark nodes and does not (easily) recover from churn. With Netvigator, a set of landmark nodes L are probed asynchronously by N nodes using Traceroute (L*N probes). Each node reports its measurements to a repository (typically a server node), which estimates a global graph of latencies. Netvigator was originally designed for proximity estimation, that is, to rank nodes according to proximity to any given node.

6 Evaluation

In order to evaluate our system, we have performed several experiments, using simulations and live tests on PlanetLab. We first evaluate the costs of using our migration prototype. Then, we look at core selection before analyzing the network estimation accuracy and its influence on the core selection.

6.1 Migration

The migration and name service are implemented as a proof-of-concept prototype. We have tested this prototype by implementing a basic protocol for group communication, with the intention of imitating the interaction (and distribution) patterns that we would expect to see in a virtual region. The test consisted of two servers (for pre- and post-migration) and several clients. All the clients joined the same communication group at the same initial server, generating random messages at irregular intervals. After a period of time, the communication group was migrated to the post-migration server. After migration, the clients seamlessly continued their interaction. The described scenario was run in excess of 100 times. Each time the clients were able to continue their interaction unhindered. As such, the system has

Invocation type	# of Invocations	Mean overhead
Remote	100,000	2.26541 *milliseconds*
Normal	100,000	0.10125 *microseconds*

Table 2 Cost of method invocation.

shown that it is capable of migrating objects, maintain references to these objects (through the name service), and have the clients reconnect to their new location.

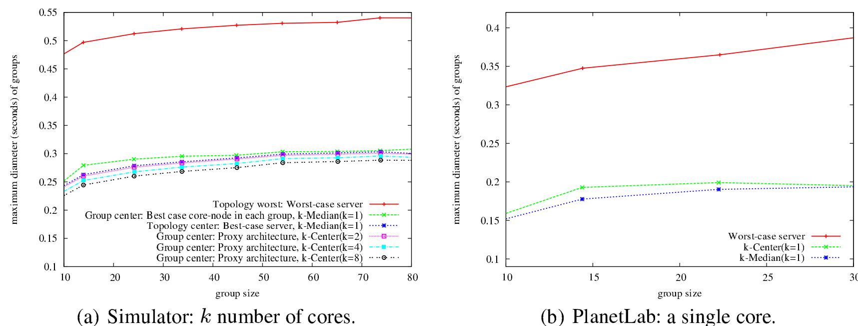
An important aspect of such a system is whether the overhead of migration is too large with respect to low latency communication. Therefore, to measure overhead, we have timed a remote method invocation (which includes a look up in the name service, serialization and deserialization, and a few other operations) and compared it to a regular method invocation. We ran the tests on an Intel Core 2 Duo, using only one core, which was clocked at 800MHz. The machine was running the operating system Linux (kernel version 2.6.22-14), and both the *sender* and *receiver* were both running on the same machine. The result is summarized in table 2, where we can see the (expected) overhead of the migration middleware itself. This overhead is considerable, but negligible when compared to the overhead added by the network. Thus, with respect to the latency gain, this is totally dependent on the core node that is found.

We also need to consider the overhead of migrating a virtual region, though this will greatly depend on the number of objects being migrated, and the size of the objects in question. With our middleware, we migrate data only, as the code is shared. We give application developers the possibility of defining the parts of an object that they wish to migrate, as detailed in section 3.3. The serialization mechanisms and more are described in further detail in [5, 6].

6.2 Core selection

Now that we are able to transparently migrate the game state to another node, we need to find the most appropriate node to migrate to. To test our proxy (core) selection algorithms, we have simulated several algorithms [43] mimicking group communication in a game, and we present the results from two of the most promising: k-median and k-center (section 4.1.1 and 4.1.2, respectively).

In our first experiment, we perform a simulation. Our network is generated using BRITE [35] with flat, undirected Waxman topologies [44] consisting of 1000 nodes. The network layout is a square world with sides equal to 200 ms. The nodes join and leave groups throughout the simulation, causing group membership to be dynamic, and group popularity is distributed according to a Zipf distribution [9]. As a metric, we use the worst-case pair-wise latency between clients (diameter) in a network (measured from the core node). Thus, the diameter should be below the latency requirements of the application (see section 2.1), and it is desirable that the diameter is as low as possible. Figure 5(a) plots the average group diameter for which a single core-node selection algorithm has chosen a server-node as the root of the group tree (all communication flows via the root). We see that choosing the server to be in the topology center does significantly reduce the group diameter. It is also clear that having a limited number of proxies (other core nodes) placed around the world can reduce the group diameter. And as expected, increasing the number of proxies does decrease the diameter further.



(a) Simulator: k number of cores.

(b) PlanetLab: a single core.

Fig. 5 Diameter (seconds) of groups using core selection.

In our next experiment, we performed experiments on PlanetLab. Figure 5(b) plots the group tree diameter based on experiments done on 100 PlanetLab nodes. We observe similar results as from the simulations using the core selection algorithms to select a server, we can greatly reduce the diameter of the communication tree.

As our results show, we are able to find a suitable proxies which lowers the diameter of the distribution tree, when taking into account the available proxies and the locations of the group of players active in the give region.

6.3 Latency estimation

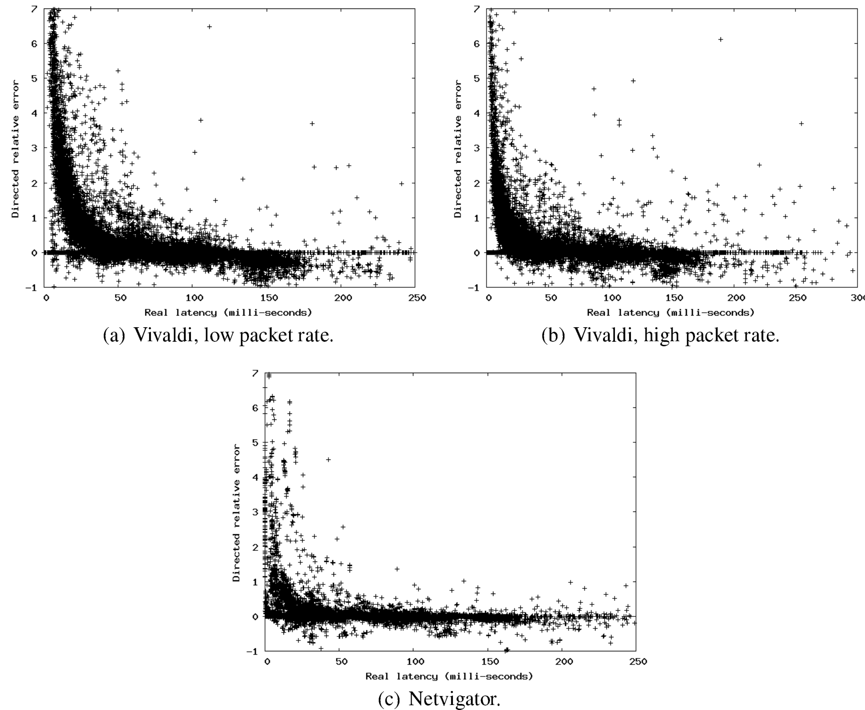
As mentioned above, in order to make an efficient server selection using the core selection algorithms, we need information about the network. In section 5, we saw that full monitoring is too expensive and estimations are therefore frequently used. In this section, we therefore present results from experiments measuring the accuracy of Netvigator and Vivaldi by comparing their estimates to real all-to-all ping measurements. Then, we look at the estimates' influence on the server (core) selection algorithms. For our experiments, we again used PlanetLab using 215 nodes (the total number of nodes we were able to access). We performed latency tests over a period of 10 days.

For the Netvigator experiments, we used publicly available estimates performed on PlanetLab. Netvigator is currently a running PlanetLab service that estimates the link latencies between nearly every PlanetLab node. We used these measurements in our experiments. The Netvigator configuration is currently a black box for us.

For the Vivaldi experiments, we used a combination of the parameters in table 3, and used *group sizes* up to 12 nodes (more neighbors makes more measurements and better estimations [19]). The RTT measurements were obtained in two different manners using *tcpinfo* or *ping* (but the results are very similar to [43] and the plots below therefore only show the tcpinfo results). The *packet rate* was varied because a higher rate follows the actual latency development more closely, while it is also consuming more bandwidth itself, at least in the active measurements. The *log times* (t) parameter determined for how long the Vivaldi information was collected until its estimations were used for identification decisions.

Descriptions	*Configurations*
Group sizes	g = 4, 8, 12 clients
RTT measures	*tcpinfo*, *ping*
Packet rates	high (100 packets/sec.), low (2 packets/sec.)
Log times	t = 4, 8, 12, 16, 20 minutes

Table 3 Vivaldi experiment configurations.



(a) Vivaldi, low packet rate.

(b) Vivaldi, high packet rate.

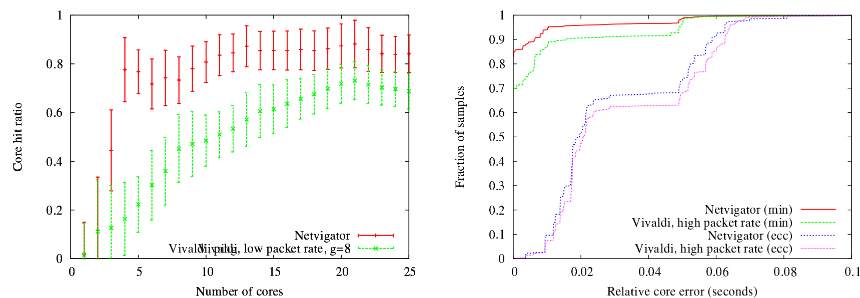
(c) Netvigator.

Fig. 6 Directed relative error of latency.

6.3.1 Estimate accuracy

As a measure of the estimation accuracy, we use the metric directed relative error, i.e., ping-measured all-to-all RTTs compared to the latency estimates for each pair of PlanetLab nodes (for other metrics, see [43]). A scatterplot of the results is shown in figure 6 plotting the directed relative deviation between the measured (real) latency versus the estimated latency, i.e., each point optimally should be on the $y = 0$ line.

We see that Netvigator is very accurate in its estimations, closely following the ideal line whereas Vivaldi has a bit more variation. Netvigator yields 80% of the estimations within a 15% relative error of the ping measurements and is clearly best. Vivaldi estimations are best in configurations with a high packet rate, and yields 80% of the estimations within a 50% relative error for a 4-minute log time. This is because more probes generates more statistics for Vivaldi to measure from in a shorter time-span. Lower packet rates require a considerably higher log time to approach the relative error satisfyingly. Furthermore, both



(a) Ratio of optimal core selection hits when k-Median is applied to Vivaldi and Netvigator estimates.

(b) CDFs of relative core error. Minimum (min) latency and eccentricity (ecc) for cores from estimates to real cores.

Fig. 7 Netvigator and Vivaldi performance in core selection.

Vivaldi, and to a lesser extent, Netvigator, overestimate RTTs for the smaller actual RTTs, while underestimating for longer distances. When the actual RTTs are very small, the overestimations are relatively high, but the absolute deviation may still be acceptable for many applications.

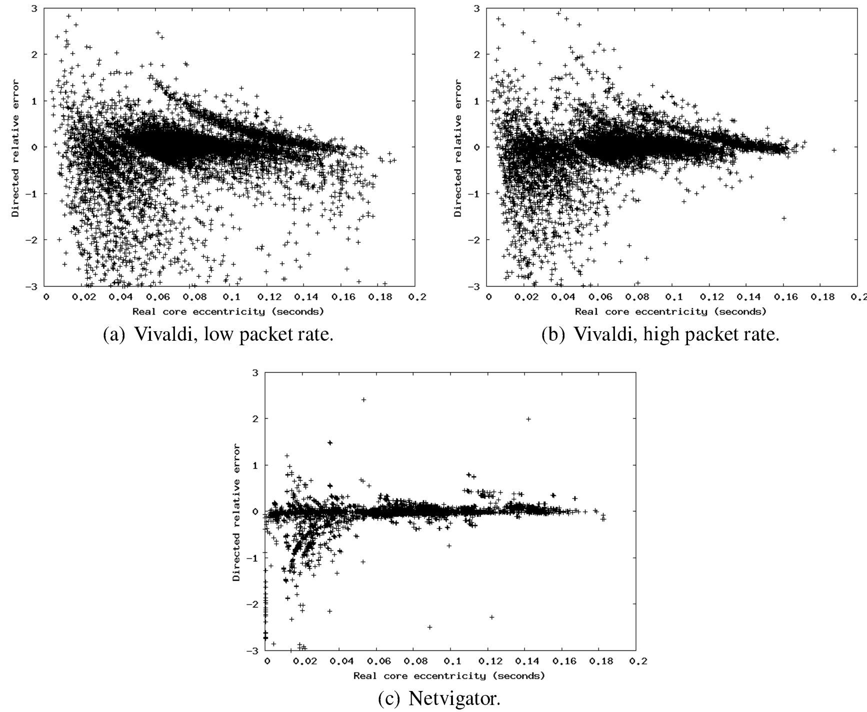
6.3.2 Latency estimates applied to core-node selection

The question now is how the estimation (in)accuracies influence the core selection algorithms. The following results are gathered by applying the Netvigator and Vivaldi latency estimates to the core-node selection algorithm k-Median, $1 < k \leq 25$. For the experiment, we used 100 nodes from PlanetLab. The core nodes were found using the network information obtained using both the estimation techniques and all-to-all measurements. For the Vivaldi estimates, we allowed a period of 4 minutes to let the node coordinates stabilize. The tests were run each day for a 10 day period.

Figure 7(a) plots the core-node selection hit ratio, which measures the ratio of "hits" for each time the k-Median finds the same core-node using estimated latencies and real all-to-all ping measurements. It is clear that Netvigator yields better estimates for use in core-node search, and stabilizes around 80% core-node selection hit ratio quickly. The hit-ratio is much lower using Vivaldi estimates (using a high packet rate gives slightly better results), especially when the number of core-nodes is less than 10.

Figure 7(b) plots the CDF of the minimum (min) core-node error in terms of latency between the core-nodes found using latency estimates and the (optimal) core-nodes found using the real all-to-all ping measurements. If the core-node error is zero, the same core-node(s) is found using estimates and real measurements. As expected, k-Median is most accurate when it uses Netvigator, with 95 % of the core-nodes within 10 milliseconds of an optimal core-node. The Vivaldi estimates makes k-Median return more inaccurate results, with 85 % of the core-nodes within 10 milliseconds of an optimal core-node. Moreover, figure 7(b) also plots the maximum latency (ecc) between the core-nodes found using latency estimates and the optimal core-nodes. Finally, the discrepancy between the reported eccentricities and the real eccentricities are plotted in figure 8 in terms of the directed relative error. Netvigator is clearly the best and has a performance very close to the real eccentricity.

In summary, it is clear that Netvigator's more accurate estimates enable the core-node selection algorithm to return the better core-nodes. Nevertheless, Vivaldi's estimates still



(a) Vivaldi, low packet rate.

(b) Vivaldi, high packet rate.

(c) Netvigator.

Fig. 8 Directed relative error of the reported eccentricity to the real eccentricity from cores to group members.

enable the core selection to find nodes within 20 milliseconds of the optimal core-nodes. Thus, both estimation techniques could be used. An important note, however, is that the performance of Vivaldi depends on the packet rate, and in the game scenario, the packet rate is closer to the tested low rate. Piggybacking Vivaldi data in the data packets might therefore not be sufficient and additional probing packets might be required.

7 Discussion

Following the discussion throughout this paper, a few things become apparent. In order to perform migration we need a way of maintaining references to objects. As such, any object is identified through the name service by querying its home-address, an identifier which is provided by the name service at object creation. The indirection through the name service is necessary for accessing objects at remote nodes transparently. Core selection is dependent on full knowledge of the network, but obtaining latency measurements through active probing is not a scalable solution. Latency estimation techniques provide an alternative.

7.1 Implications of migration

Due to the distributed name service resolving references to remote objects can become a time consuming task. An object that is frequently migrated will create trails of object references

at the nodes it visits. To reach the object, one could be required to unravel a number of these references before being able to access the object, though this will depend on where in the chain of references the object request propagates from. One possible solution to solving this problem is to use leases, as originally described by Gray et al [28]. They describe a lease as a promise from the server that it will push updates to the client for a specified amount of time. A more flexible approach is described by Duvvuri et al [22] in form of adaptive leases, which make it possible for the server to adapt leases depending on different criteria. Thus, the node that has migrated the object can request a lease on an object, and have the server send it updates about the object reference for the duration of that lease.

Another concern is related to the side-effects of migrating game-state during execution of the simulation. If migration is not performed transparently, we might adversely affect the interacting players. In [36], Nelson et al demonstrate how an active application is moved from one virtual machine to another, with minimal perceived impact to the user's interaction with the application.

We also need to consider how to clean-up objects that are no longer in use. With a distributed system, where objects are frequently moved around, the garbage collection process becomes more complicated then otherwise. One solution is to use reference counting, when the number of references to an object reaches zero, we can remove the object. As such, each object holding a reference to another object must send a message when it is no longer interested in holding a reference.

7.2 Partial failures

Partial failures occur when a node in a distributed system becomes unavailable, effectively rendering the objects managed by it inaccessible. The current architecture does not accommodate for partial failures, but there are ways to minimize the repercussions of these incidents. One possibility has its roots in peer to peer based file systems, where copies of an object will be distributed to several nodes in the system. PAST [40] and OceanStore [27] have, for example, implemented such systems with success. PAST copies objects to random nodes, in an attempt to distribute the objects evenly. OceanStore uses a more deterministic approach, and places the objects close to nodes which access them. Lookup of objects in the system can then be implemented in a fashion similar to that of Chord [42] or Tapestry [46]. These implementations are based on the principle of incrementally forwarding messages from point to point, until they reach their destination. Each node in the system keeps a small routing map, which is used to determine which nodes to forward the message to. A problem with this type of look up is that the response time might be too high for interactive applications. A centralized approach would avoid the lookup time problem, but would itself become a point of failure, and would also have to handle all the network messages, which could potentially congest the server.

7.3 Latency estimation techniques

We have evaluated Netvigator and Vivaldi as popular representatives for the latency estimation techniques. It is clear that Netvigator yields better estimations, but it is also more difficult to set up as it dependens on landmark nodes. In the scenario of MMOGs, when the game provider has dedicated servers, this is not an issue, the benefit of the increased accuracy outweighs the initial setup cost. However, while Vivaldi does perform worse, it

has the advantage of easy deployment and the ability to recover from churn. The Netvigator configuration is a blackbox, as PlanetLab has not disclosed this information. For Vivaldi, however, the best configuration was a group size of 8 (and above) and high packet rates. A low packet rate reduced the estimation accuracy, and required 8 minutes to stabilize, in contrast to 4 minutes at high packet rates. This might be important to keep in mind, as Vivaldi piggybacks probing information, and many games have a low packet rate [38], i.e., additional probing packets might be needed using Vivaldi. For both Netvigator and Vivaldi, k-Median is able to find close-to-optimal core-nodes, and it is clear that Netvigator estimates enable the core-node selection algorithm to return the more optimal core-nodes. Nevertheless, Vivaldi's estimates still enable the core selection to find nodes within some tens of milliseconds distance of the optimal core-nodes (with the server densities that we tested). Thus, both estimation techniques could be used.

7.4 Activation policies

Core selection and migration require resources in the form of processing power and network messages when activated. As such, they should be triggered methodically and on-demand, with policies governing their activation. The combined process consists of two stages, where the first stage evaluates if activating a core selection is potentially beneficial. The activation criteria we have identified so far are: churn (in the form of players joining/leaving), time of day, player arrival rate, history based (this being a preemptive approach, where known situations, such as battles, trigger the activation) and server-load. If the first stage is completed successfully, and a new core is selected, the second stage needs to determine if performing a migration is beneficial. Metrics of consideration include: threshold of aggregate latency, number of players, packet loss, and server load. For both stages the metrics are not necessarily mutually exclusive, and we do not consider these lists to be exhaustive.

8 Conclusion

It has been shown that there is a strong correlation between latency and the playability of an online game [17], with the perceived game play deteriorating considerably as the latency increases.

An important factor for world-spanning games, such as MMOGs, lies in the diversity of its user base. There will, at any point in time, be a number of players connected from different physical locations. It has, however, been shown that distinct groupings will appear, and change with the time-of-day [25, 29]. There have been made few efforts into determining how to best support the dynamic player masses in virtual worlds hosting thousands of concurrently interacting players, when geographically distributed servers are available.

In this paper, we have presented a viable solution with geographically distributed servers. We have shown how core selection can be used to find an optimal node in the system for placing a virtual region, and correspondingly the players interacting in that region. We have looked at how latency estimation techniques can be utilized to gather information about the network in a scalable manner. Once an optimal node has been located we can migrate the game state to that node, maintaining references to the migrated state through the use of our distributed name service. By performing this migration, the overall latency of that region can be lowered decreasing the average network latency.

Thus far, we have implemented the migration and name service as a proof-of-concept, and run some basic tests to determine its usability. Furthermore, we have run simulations on core selection and determined its applicability in the scenarios we have described. Additionally, we have run extensive tests that show how latency estimation, using for example Vivaldi or Netvigator, provides a scalable approach to gathering information about the network. Now, it remains to integrate this functionality with an application and run large-scale tests.

References

1. ARMITAGE, G. Client-Side Adaptive Search Optimisation for Online Game Server Discovery. *Lecture notes in computer science 4982* (2008), 494.
2. ARMITAGE, G. Optimising online fps game server discovery through clustering servers by origin autonomous system. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (May 2008).
3. BARAK, A., GUDAY, S., AND WHEELER, R. G. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
4. BEIGBEDER, T., COUGHLAN, R., LUSHER, C., PLUNKETT, J., AGU, E., AND CLAYPOOL, M. The effects of loss and latency on user performance in unreal tournament 2003. *In the Proceedings of NetGames'04, Portland, Oregon, USA* (August 2004), 144–151.
5. BESKOW, P. Migration of objects in a middleware for distributed real-time interatctive applications. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2007.
6. BESKOW, P., HALVORSEN, P., AND GRIWODZ, C. Latency reduction in massively multi-player online games by partial migration of game state. *Second International Conference on Internet Technologies and Applications, Wrexham, Wales* (September 2007), 153–163.
7. BESKOW, P., VIK, K.-H., GRIWODZ, C., AND HALVORSEN, P. Latency reduction by dynamic core selection and partial migration of game state. *Proceedings of NetGames'08, Worcester, MA, USA* (oct 2008).
8. BLAIR, G., COULSON, G., ROBIN, P., AND PAPATHOMAS, M. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (1998), Springer-Verlag, pp. 191–206.
9. BROOKES, B. The derivation and application of the Bradford-Zipf distribution. *Journal of Documentation 24*, 4 (1968), 247–265.
10. BRUN, J., SAFAEI, F., AND BOUSTEAD, P. Server topology considerations in online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2006), ACM, p. 26.
11. C. CHAMBERS, W. FENG, W. F., AND SAHA, D. A geographic redirection service for on-line games. *In Proceedings of the eleventh ACM international conference on Multimedia, Berkeley, CA, USA* (November 2003), 227–230.
12. CHAMBERS, C., CHANG FENG, W., SAHU, S., AND SAHA, D. Measurement-based characterization of a collection of on-line games. *In the Proceedings of the 5th ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA* (October 2005), 1–14.
13. CHANG FENG, W., CHANG, F., CHI FENG, W., AND WALPOLE, J. Provisioning on-line games: a traffic analysis of a busy Counter-strike server. *In the Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement, Marseille, France* (November 2002), 151–156.
14. CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 273–286.
15. CLAYPOOL, M. The effect of latency on user performance in real-time strategy games. *Elsevier Computer Networks 49*, 1 (Sept. 2005), 52–70.
16. CLAYPOOL, M. Network characteristics for server selection in online games. *In Proceedings of the fifteenth Annual Multimedia Computing and Networking (MMCN'08), San Jose, CA, USA 6818* (January 2008), 681808.
17. CLAYPOOL, M., AND CLAYPOOL, K. Latency and player actions in online games. *Communications of the ACM 49*, 11 (Nov. 2005), 40–45.

18. Costa, M., Castro, M., Rowstron, A., and Key, P. Pic: Practical internet coordinates for distance estimation. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 178–187.
19. Dabek, F., Cox, R., Kaashoek, F., and Morris, R. Vivaldi: a decentralized network coordinate system. In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2004), pp. 15–26.
20. Dick, M., Wellnitz, O., and Wolf, L. Analysis of factors affecting players' performance and perception in multiplayer games. *In the Proceedings of NetGames'05, Hawthorne, NY, USA* (October 2005), 1–7.
21. Dovrolis, C., Ramanathan, P., and Moore, D. What do packet dispersion techniques measure? In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2001), vol. 2.
22. Duvvuri, V., Shenoy, P., and Tewari, R. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Transactions on Knowledge and Data Engineering 15*, 5 (2003), 1266–1276.
23. Egenhofer, M., and Spatial, S. A Query and Presentation Language. *IEEE Transactions on Knowledge and Data Engineering 6*, 1 (1994), 86–95.
24. Elmokashfi, A., kleis, M., and Popescu, A. Netforecast: A delay prediction scheme for provider controlled networks. In *IEEE Globecom* (November 2007).
25. Feng, W., and Feng, W. On the geographic distribution of on-line game servers and players. *In the Proceedings of NetGames'03, Redwood City, California, USA* (May 2003), 173–179.
26. Funcom. Anarchy Online. *http://www.anarchy-online.com/, June* (2008).
27. Geels, D. Data Replication in OceanStore. Tech. Rep. UCB//CSD-02-1217, Computer Science Division, U. C. Berkeley, nov 2002.
28. Gray, C., and Cheriton, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev. 23*, 5 (1989), 202–210.
29. Griwodz, C., and Halvorsen, P. The fun of using TCP for an MMORPG. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (May 2006), ACM Press, pp. 1–7.
30. Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems (TOCS) 6*, 1 (1988), 109–133.
31. Karaman, A., and Hassanein, H. S. Core-selection algorithms in multicast routing - comparative and complexity analysis. *Computer Communications 29*, 8 (2006), 998–1014.
32. King, B. GCC-XML the xml output extension to gcc. *Undated. Online: http://www. gccxml. org/HTML/Index. html.*
33. Kupfer, M. D. Sprite on mach. In *MSYM'93: Proceedings of the 3rd conference on USENIX MACH III Symposium* (Berkeley, CA, USA, 1993), USENIX Association, pp. 7–7.
34. Lee, K., Ko, B., and Calo, S. Adaptive server selection for large scale interactive online games. *Computer Networks 49*, 1 (2005), 84–102.
35. Medina, A., Lakhina, A., Matta, I., and Byers, J. BRITE: Universal topology generation from a user's perspective. Tech. Rep. BUCS-TR-2001-003, Computer Science Department, Boston University, Apr. 2001.
36. Nelson, M., Lim, B., and Hutchins, G. Fast transparent migration for virtual machines. *Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference table of contents* (2005), 25–25.
37. Ng, T., and Zhang, H. Towards global network positioning. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement* (2001), ACM New York, NY, USA, pp. 25–29.
38. Petlund, A., Evensen, K., , Griwodz, C., and Halvorsen, P. Improving application layer latency for reliable thin-stream game traffic. In *Workshop on Network and System Support for Games (NETGAMES)* (Oct. 2008), pp. 91–98.
39. Powell, M. L., and Miller, B. P. Process migration in demos/mp. Tech. rep., Berkeley, CA, USA, 1983.
40. Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of SOSP'01, Lake Louise, Alberta, Canada* (oct 2001), 188–201.
41. Sharma, P., Xu, Z., Banerjee, S., and Lee, S.-J. Estimating network proximity and latency. *SIGCOMM Comput. Commun. Rev. 36*, 3 (2006), 39–50.
42. Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of SIGCOMM' 01, San Diego, CA, USA* (aug 2001), 149–160.
43. Vik, K.-H. *Group Communication Techniques in Overlay Networks (submitted).* PhD thesis, Department of Informatics, University of Oslo, Norway, Dec. 2008.

44. WAXMAN, B. M. Dynamic Steiner tree problem. *SIAM J. Discrete Math. 4* (1991), 364–384.

45. WONG, B., SLIVKINS, A., AND SIRER, E. Meridian: A lightweight network location service without virtual coordinates, 2005.

46. ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, apr 2001.

47. ZNATI, T. B., AND MOLKA, J. A simulation based analysis of naming schemes for distributed systems. In *Proceedings of the 25th Annual Simulation Symposium* (Los Alamitos, CA, USA, Apr. 1992), pp. 42–53.

Paul Beskow received his B.Sc. in Informatics in 2003, and his M.Sc. in 2007, both at the University of Oslo (UiO) and is currently working on his Ph.D. at the same institution. His main research interests include system support for low-latency communication including object migration, network protocol optimizations, as well as various TCP protocol optimization issues.

Knut-Helge Vik is a Ph.D. student at the Simula Research Laboratory. His recent research activities address system demands for distributed interactive applications. The research include graph theoretical problems for overlay network design, practical simulations and experiments of overlay network construction algorithms, group management requirements related to latency and consistency, etc. He received a Master of Science in Computer Science from Washington State University in 2004, and has been a Ph.D. student at the Department of Informatics, University of Oslo and Simula Research Laboratory since October 2004. More information and the publication list can be found at http://home.ifi.uio.no/ knuthelv.

Dr. Ing. Carsten Griwodz is an associate professor at Simula Research Laboratory, Norway and at the Department of Informatics at the University of Oslo. He is member of the Center for Research-based Innovation "Information Access Disruptions". He received his Diploma in Computer Science from the University of Paderborn, Germany, in 1993. From 1993 to 1997, he worked at the IBM European Networking Center in Heidelberg, Germany. In 1997 he joined the multimedia communications lab at Darmstadt University of Technology, Germany, where he obtained his doctoral degree in 2000. His interests lie in the improvement of system support for interactive distributed multimedia, with operating systems and protocol support for on-demand streaming applications and multiplayer games in particular.

Dr. Scient. Pål Halvorsen is an associate professor at Simula Research Laboratory and at the Department of Informatics, University of Oslo, Norway. He is member of the Center for Research-based Innovation "Information Access Disruptions". He received his master and doctoral degree in computer science from the University of Oslo in 1997 and 2001, respectively. His research activities focus mostly on resource utilization and system support for distributed multimedia systems, and in particular, in the area of on-demand streaming applications and multiplayer games.

Paper V

Reducing game latency by migration, core-selection and TCP modifications

Paul B. Beskow, Andreas Petlund, Geir A. Erikstad, Carsten Griwodzand Pål Halvorsen

Published: Inderscience International Journal of Advanced Media and Communication (IJAMC) 4(4):343-363, 2010

Evaluation: This journal paper was submitted by invitation to a Special Issue of the International Journal of Advanced Media and Communication (IJAMC).

Author contribution: Beskow was the primary contributor to this article, implementing application code and running tests. Petlund provided knowledge on thin-streams and mechanisms for adaptation.

Reducing game latency by migration, core-selection and TCP modifications

Paul B. Beskow, Andreas Petlund, Geir A. Erikstad, Carsten Griwodz, Pål Halvorsen

Simula Research Laboratory, N-1325 Lysaker, Norway
Department of Informatics, Univ. Oslo, N-0316 Oslo, Norway
Email: {paulbb, apetlund, geirerik, griff, paalh}@ifi.uio.no

Abstract: Massively multi-player online games (MMOGs) have stringent latency requirements and handle large numbers of concurrent players. To support these conflicting requirements, it is common to divide the virtual environment into virtual regions, and spawn multiple instances of isolated game areas; to serve multiple distinct groups of players. As MMOGs attract players across the world, dispersing these regions and instances on geographically distributed servers is plausible. With Ginnungagap, we present the prototype implementation and evaluation of functionality for migrating objects to support dynamic re-distribution of game state; with a distributed name server efficiently maintaining references to migrated objects. Core selection algorithms, using players latencies, can then locate an optimal placement for a given region or instance. The variance of the measured latencies (and core selection algorithm) are reduced by enabling our TCP modifications for latency reduction for non-greedy streams. Correspondingly, we anticipate a decrease in aggregate latency for the affected players.

Keywords: Latency reduction, massively multi-player online games, TCP enhancements, core selection algorithms, object migration, distributed name server

Biographical notes: Paul B. Beskow is a PhD student at the Department for Informatics, University of Oslo and Simula Research Laboratory. His research interests include optimization of resource utilization and distributed processing.
Andreas Petlund is a PostDoc at the Department of Informatics, University of Oslo and at Simula Research Laboratory. His research interests include network protocol optimization for time-dependant thin streams, operating systems optimisations and hardware offloading.
Geir A. Erikstad is a former master student at the Department for Informatics, University of Oslo. His main research interests include game latency optimisations.
Carsten Griwodz is a Professor at the Department of Informatics, University of Oslo and a researcher at Simula Research Laboratory. His research focuses mainly on distributed multimedia systems.
Pål Halvorsen is an Associate Professor at the Department of Informatics, University of Oslo and a researcher at Simula Research Laboratory. His research focuses mainly on distributed multimedia systems.

Copyright © 2009 Inderscience Enterprises Ltd.

1 Introduction

In online games, some latency is tolerable (Claypool, 2005) as long as it does not exceed the threshold for playability that ranges from 100ms to 1000ms depending on the type of game (Claypool et al., 2006), but high pair-wise latency remains a primary obstacle for the quality of perceived game-play in a number of online games. For example, Massively Multi-Player Online Games (MMOGs) often rely on a client-server model for event distribution, where the events are collected at the server, and distributed to the interacting players. Players with high latency will inadvertently have a negative effect on the perceived quality of game play (Claypool, 2005). This occurs if one or more players connections are comparatively slow, because any added delay is not isolated to the player alone. Due to a centralized server, the delay will propagate to the interacting parties and potentially result in inconsistencies, from which the server must then recover. While having minor effects on the outcome of the game, it results in a perceived deterioration to the quality of interaction (Dick et al., 2005). As such, *low latency for all interacting players* is a prevalent goal.

This goal, however, contradicts other observations of online games, such that the game traffic varies strongly with time and the attractiveness of the individual game (Feng et al., 2002; Chambers et al., 2005) and that geographical dispersion of players in an online game depends heavily on the time of day (Feng et al., 2003). Thus, in large games like Anarchy Online and EVE Online, there are always players all around the world interacting in the same game instance, although the geographical location of the large bulk of users shifts dynamically.

In Beskow et al. (2008), we proposed the initial design of a middleware supporting migration of partial game-state to servers that are dynamically selected according to the majority of the players locations. The level of game-state granularity can range from entire virtual regions, to instances, or single objects in the virtual world. With GINNUNGAGAP (in Norse mythology GINNUNGAGAP refers to the vast, primordial void that existed prior to the creation of the universe) (Beskow et al., 2009a), we present an implementation of this middleware (the source code of the middleware, and logs from the PlanetLab experiments are available for download at http://simula.no/research/networks/software/). This middleware selects servers based on maximum latencies between all the players in a region and potential servers, and if the potential gain in terms of reduced latency exceeds a given threshold, the game state is migrated to this server. In this paper, we have focused on MMOGs that use a client-server model, though GINNUNGAGAP could easily be adapted for use in peer-to-peer scenarios. In which case, all interacting parties could potentially contribute as a server, greatly increasing the probability of locating an optimal node. It is nonetheless vital that the variance of the measured latencies, provided as input for the core selection algorithm, are as low as possible; one way to ensure this is to perform a number of probes, which can be costly and time-consuming (considering the number of potential clients). Another option is to increase the resilience of single measurements by applying our thin-stream TCP modifications (Petlund et al., 2008). Thus, we present results from experiments using this middleware running a simple game both in a lab environment and on PlanetLab. In summary, we show that the dynamic selection of servers is beneficial and can benefit from our TCP modifications, and that the

overhead of migration and method invocation does not exceed the limits of a good perceived game play.

2 Background and Related Work

Migration techniques: Migration provides functionality to move entities of varying form and size (e.g., virtual machines, processes, data, code, etc.) between servers/proxies/clients in a distributed system. In Nelson et al. (2005), a virtual machine is transparently migrated with minimal impact on the user. Sprite and Mosix are *NIX based operating systems that allow for processes and their associated state to be migrated. Code migration makes it possible to defer the execution of code to an interacting party. Interpreted languages are typical for this, such as JavaScript in modern browsers and SQL in database management systems. Emerald and Chorus/COOL enable migration of objects (e.g., their code and state) during run-time execution of a program (per the object-oriented paradigm). Finally, we have eXternal Data Representation (XDR) and Boost::Serialization that provide functionality to migrate data. This is achieved by serialization, i.e., creating a binary representation of data in an application (e.g., ints, floats, chars, etc.), which is moved between instances of applications. Migration, at its various granularity levels, serves several purposes, such as dynamic load distribution, fault resilience, increasing resource locality or to facilitate system administration. We wish to achieve a combination of load distribution and increased resource locality by migrating game state to an optimal location (relative to the interacting users). Finding the correct level of granularity is important, and in online games there are two essential characteristics, 1) all code is shared, and 2) not all state related to an object needs to be migrated. Thus, we can eliminate some overhead by serializing only parts of an object. To this end, data migration accommodates both of these characteristics.
Server selection: Game server selection is an important facet of the playability for several types of online games. This is particularly true for games that are highly sensitive to latency, such as first person shooter (FPS) games. As such, the player's selection process is commonly guided by measuring dimensions that affect playability, such as latency and packet loss. This sensitivity to latency is commonly alleviated by distributing servers widely. With respect to this, Chambers et al. (2003) have looked at how server selection can be optimized for a single client, when given a set of available servers. In a further study, Claypool (2008) notes that we regularly find groups of players that wish to play together on a server, such as friends or clans (organized players). As such, he has investigated how server selection can be optimized from the perspective of a group of players. In two related studies by Armitage (2008b,a), efficient ways of ranking servers in the discovery process itself are examined. These papers have in common that they consider server selection from the perspective of the player(s), and additionally assume a certain availability of servers (it is common for geographically coupled players, such as real life friends, to play against each other). For a world spanning game, however, where all users interact in the same game instance, such as an MMOG, the number of available servers is often limited. Lee et al. (2005) present their heuristic for selecting a minimum number of servers satisfying given delay

constraints (from the perspective of large scale interactive online games, such as MMOGs). Their aim, however, is to have well provisioned network paths in a centralized architecture. Thus, they do not consider the aspect of geographical dispersion of players. In a similar study, Brun et al. (2006) investigate how a server's location can influence the fairness of a game, and how selecting an appropriate server impacts this fairness. They use an objective function, which they call *critical response time* to rank the servers.

3 Implementation of Ginnungagap

The development of Ginnungagap has been driven by the motivation of lowering the aggregate latency for groups of interacting players in interactive online games, such as MMOGs. To achieve this, three components were necessary, 1) a way of locating a server or proxy closer to the center of the players through core-selection, 2) a means of facilitating the migration of game-state, i.e., re-locating the players and the objects they interact with, and 3) allow for continuous interaction with the virtual environment, through remote method invocations (RMI), even during migration. Ginnungagap supports all three components.

At its core Ginnungagap runs three threads, **send**, which is responsible for transmitting middleware messages, labeled either RMI or MIG (method invocation and migration, respectively, see tables 1 and 2); **receive**, which accepts new connections and receives transmitted messages; and **process**, which processes the messages, and activates core-selection at given intervals in the servers and proxies. In addition, all clients run a (application level) **ping** thread, which is responsible for gathering and transmitting its latency information to the proxies and servers, which is used as input to the core-selection algorithm (explained in section 3.1).

All objects that support migration and RMI inherit from a base class that provides a minimal interface of methods and data that, correspondingly, must be implemented and present in all inheriting classes. The essential data in an object consists of the object's mode and its identifier. The mode of the object can be either `MIGRATING`, which implies that the object is in transit, and RMIs must be buffered until they can be forwarded and processed, or `NORMAL`, which implies that an RMI can be processed immediately. As an alternative to buffering messages it could be more beneficial to utilize related ideas, such as transparent migration of virtual machines (Nelson et al., 2005), where one variation utilizes a two-stage approach, with an initial push-phase, where data is pushed to the receiving node, data modified during this stage is marked as dirty and is retransmitted during the stop-and-copy phase, where, in our case, the object is again made available for interaction. The unique identifier of an object follows it throughout its life-time and throughout the distributed system, and is used to uniquely locate that object at any node. A name service is necessary, however, to maintain knowledge of which node the object is currently residing at.

Instrumental to the name service is an efficient way of maintaining references to the objects as they migrate. To accomplish this we have implemented a distributed name service. A name service can be implemented in several ways, which we discuss in the context of MMOGs in Beskow (2007). The conclusion is that a distributed name service is an efficient way of handling references as there is a

minimal overhead in binding an object with the name service, because each node has its own name service, where the objects are bound initially. Communication between nodes (and thus the name services) only becomes necessary when an object is migrated. Given that servers in the system are geographically distributed, binding objects locally becomes quite beneficial. One other advantage is that there is no single point of failure, so large parts of the application can continue running if a server fails. Look-ups are also efficient, as we can directly query the node our name service has registered as current caretaker of the object. This access time, however, depends on the number of times an object has been migrated (after its point of creation) and at which point in this chain the invocation is performed. We partially alleviate this situation by embedding the calling nodes' information in a message that passes through such a chain, such that the reply is sent directly back to the calling node.

3.1 Core selection

Vik (2009) identified k-Median (see algorithm 1) as the heuristic algorithm for the graph-theoretical problem of locating an optimal proxy for a set of interacting clients, and this is the algorithm we use in the core-selection component. However, any of the node selection algorithms presented by Vik can be used instead of the k-Median algorithm in our middleware.

Algorithm 1 k-MEDIAN(G):

1: Input: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
2: Output: A set $C \in X$ of core-nodes.
3: map<x-id, pair-wise> mapIdPairwise
4: **for** each $x \in X$ **do**
5: pairwise = getPairwiseDistances(x, Z)
6: mapIdPairwise.insert(x-id, pairwise)
7: **end for**
8: C = kLowestPairwise(mapIdPairwise)

The k-Median core-node selection algorithm finds k core-nodes that are the k nodes with the lowest average pair-wise distances to the nodes in the member-node set. The algorithm solves the k-minimum-pairwise problem, which when given a weighted graph $G = (V, E, c)$ and an integer $0 < k < |V|$, finds a set $C \subset V$ of size k, such that the sum of the distances from the vertices's $u \in C$ to all nodes $v \in V$ is minimal. The k-Median algorithm has a time-complexity of $O(n^2)$ on any graph.

3.2 Migration

Migration (see figure 1 for an overview of the process) targets are found by core-selection **(1)**, though migration is only performed when the latency gain increases above a predefined threshold. The migration is performed by the `MigrationService` **(2)**, which first sets the mode of the migrating object to `MIGRATING` **(3)**. Thus, all incoming RMIs are temporarily buffered locally, until they are forwarded and processed by the object at its new location. The buffered RMIs are marked as chained invocations, and as such, contain information about the node from which the call originated.

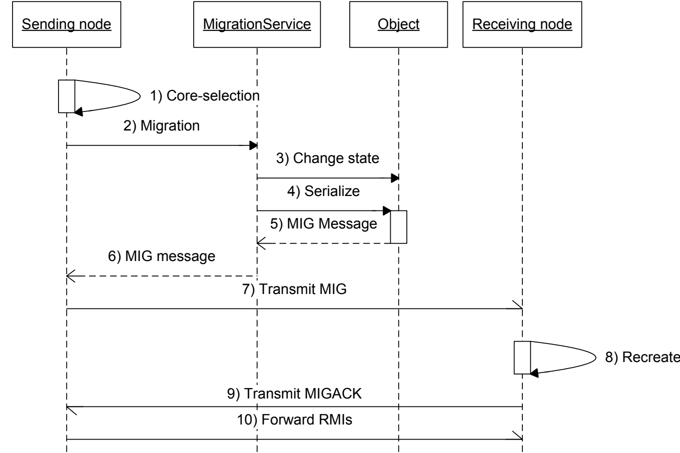


Figure 1 Sequence of a migration

After the mode update, the object is serialized **(4)**. The serialization results in the creation of a `MIG` message (see table 1) in the binary XDR format **(5)**. Not all attributes of an object are necessarily serialized; only those marked for serialization by the programmer. Once the message has been packed it is transmitted to the receiving node **(7)**.

Message type	Object id	Object type	Attribute	Attribute
MIG	7e2...31f	CLASS_PLAYER	Odin	98

Table 1 Migration message

At the receiving node, the message is processed **(8)**. The middleware reads the type of the message that it has received (`MIG`) and invokes the appropriate handler. It passes the remainder of the message to the MIG-handler, which reads the object type and requests that a new object instance of that type is created. The instance is created by calling the object constructor, which initializes the object with the attributes deserialized from the message.

Once the object has been created and initialized, the MIG-handler creates a MIGACK message with the id of the object and transmits this message to the sending node **(9)**. Upon receiving this message, the sending node knows that the migration has been completed and forwards any waiting RMI-messages in its buffer **(10)**.

This same process is used to migrate groups of objects, though with a slight modification. In the case of groups, all the objects are serialized into one large message, instead of being sent in small individual ones.

3.3 Remote method invocation

An RMI is the invocation of a method on an object that is not residing in local memory (see figure 2 for an overview). To accomplish this goal, a number of components need to be present in the middleware. In this example, we detail the interaction of these components, starting with a player logging into a remote server:

```
dist_ptr<Player> plyr = Startup::login(‘‘Odin’’);
```

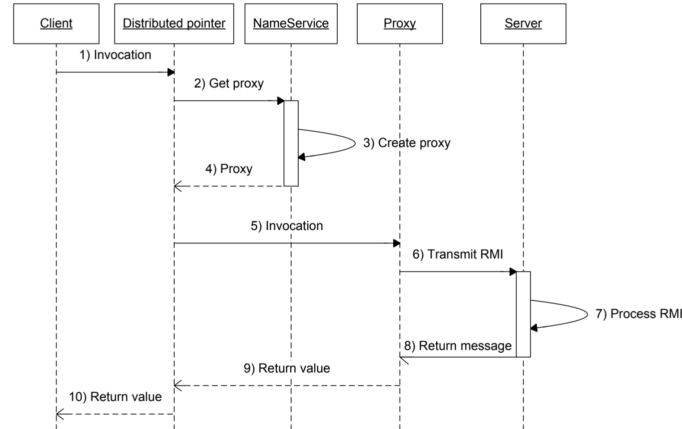


Figure 2 Sequence of a RMI

This `login(...)` invocation is in itself an RMI, but is possible because the server and `Startup` object have a well-known location and identifier. With this invocation, the server creates a player object and returns its corresponding universally unique identifier (UUID), which is guaranteed unique for this object throughout its lifetime. The UUID is entered into the name service of the local node. Together with network information about the sending node, this ensures that we have a way of contacting the remote node and identifying that instance of the object class. The UUID is also stored in the distributed pointer `plyr`. After input from the player, a request to move is issued:

```
plyr->move( Direction, Distance );
```

As `plyr` is a distributed pointer, the `move` method invocation is intercepted by the smart pointer **(1)**, which contains the UUID of the object it points to and is used to query the name service for a pointer to the object corresponding to that UUID. The name service looks it up in its records and performs one of two actions, 1) returns a pointer to the local object, or 2) returns a pointer to a proxy object if it is remote **(2)**. At the first request for a remote object, a proxy is created **(3)**, and is reused for subsequent requests.

As the object we have invoked is not local, the distributed pointer redirects the call to the move-method through the proxy object with the parameters passed to the method **(5)**. Methods in an object that support RMI are implemented as virtual methods and have their own implementation in the proxy object.

At this point the proxy object assumes control, and creates a message of type RMI (see table 2). This identifies the class type of the object, the specific object (UUID) and the function that is being invoked, and its corresponding parameters. All this information is serialized using XDR. With knowledge that the object is remote, the proxy object then transmits the message to its destination **(6)**.

Message type	Object id	Object type	Method	Parameter	Parameter
RMI	5c1...13c	CLASS_PLAYER	METH_MOVE	North	50

Table 2 RMI message

At the receiving node the message is processed by the middleware **(7)**. It determines the message type (`RMI`), and invokes its corresponding handler. This

RMI-handler determines if the object is local or remote, and in the latter case passes the message on (with the information of the emanating node embedded in the message). Thus, the reply, once the RMI is processed, can be sent directly to the original caller, effectively minimizing extraneous messages in the network. If the object is local, the corresponding skeleton method is invoked, which deserializes the parameters of the object, and invokes the method of the local object. Once the invoked method returns, the RMI has completed successfully at the remote node, and a potential return value from the function itself is serialized and returned. The middleware itself generates a reply to the calling node with information of the successful completion **(8)**. Finally, this return value is processed by the calling node **(10)**.

4 TCP enhancements for thin streams

TCP is used as the primary transport protocol in a number of existing online games (including MMOGs), because it inherently provides reliability and simplifies firewall traversal. Though using TCP in this scenario comes at a cost due to TCP's inherent retransmission mechanisms, which are optimized for TCP's targeted audience; bulk transfer scenarios. This is in contrast to the network traffic patterns exhibited by a number of online games, which send small packets with high inter-arrival times (*thin streams*). As a result, supporting MMOGs with reliable, low-rate, interactive streams using TCP on the Internet poses huge challenges due to packet loss. To reduce the loss-recovery delays, especially the infrequent but worst-case scenarios that have devastating impact on QoE, and to minimize the variance in the latency estimates for server selection, we have implemented enhancements to TCP's retransmission policies to reduce the latencies for thin streams (Petlund et al., 2008; Petlund, 2009). These modifications only need to be present at the sender (though the benefit is greater if both sides of the connection have them), and, as such, we support unmodified receivers running commodity operating systems (Linux, *BSD, OS X, Windows, etc.). These modifications are backwards compatible and only triggered when the system detects the described thin stream properties. In particular, we have changed the exponential back-off and fast retransmit algorithms, and added a bundling mechanism that uses available TCP segments to resend unacknowledged application data. It is important to acknowledge that these modifications are only enabled by the kernel for thin-streams, since enabling these mechanisms on greedy streams would violate TCP fairness principles.

4.1 Modified exponential back-off

TCP retransmissions are triggered by timeouts when a segment is not acknowledged in time and no fast retransmit occurs. To avoid congestion in a scenario with multiple consecutive losses, the retransmission timer is doubled each time (exponential back-off in figure 3). This leads to retransmission delays that increase exponentially when multiple consecutive packet losses occur. However, in the MMOG scenario, it is not necessary to use exponential back-off to prevent aggressive probing for bandwidth, because the stream is thin.

We therefore use linear timeouts for the first 6 timeouts, as shown in figure 3, before we switch to exponential back-off. For further discussions on the impact of these modifications on fairness and friendliness, we refer to Petlund (2009).

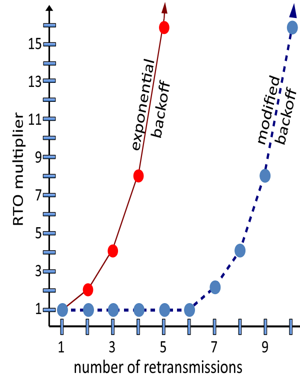
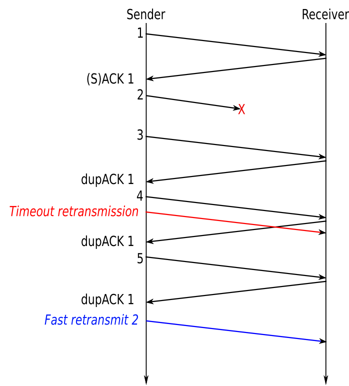


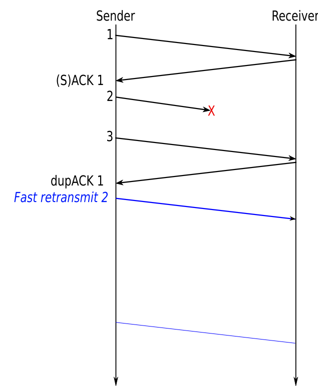
Figure 3 Modified vs. exponential back-off

4.2 Modified fast retransmit

Due to the high packet interarrival time in the game scenario, fast retransmissions occur seldom. In most cases, a retransmission by timeout is triggered before the three (S)ACKs required to trigger a fast retransmission are received (figure 4(a)). To deal with this problem, we allow a fast retransmission to be triggered by the first indication that a packet is lost (figure 4(b)). This is because retransmissions triggered by causes other than timeouts are usually preferable with respect to latency (Petlund, 2009).



(a) Timeout and fast retransmission

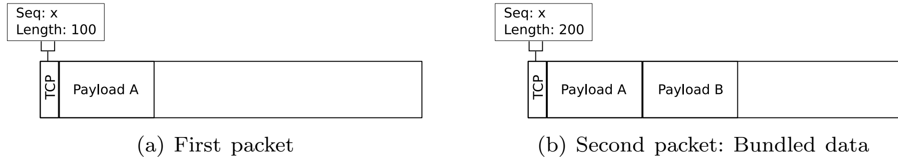


(b) Modified fast retransmission

Figure 4 Fast retransmissions

4.3 Redundant data bundling

To avoid retransmissions, we redundantly copy (bundle) data from unacknowledged messages in the send queue into the new packet as long as the combined packet size is less than the maximum segment size (MSS) (to avoid IP fragmentation). If a retransmission occurs, as many of the remaining unacknowledged packets as possible are bundled with the retransmission. This increases the probability that a lost segment is recovered faster. Figure 5 shows how a previously transmitted data segment is bundled with the next packet. The sequence number stays the same while the packet length is increased. If packet A is lost, the ACK from packet B acknowledges both segments, making a retransmission unnecessary. In contrast to the modified retransmission mechanisms of exponential back-off and fast retransmit, which are triggered when fewer than



(a) First packet (b) Second packet: Bundled data

Figure 5 Bundling unacknowledged data

four packets are in transit, redundant data bundling (RDB) is enabled by small packet sizes, and limited by the packet inter-arrival times (IATs) of the stream. If the segment is filled up to MTU size, either due to large writes from user space or due to sufficient data in the TCP send queue, which is typical for bulk data transfer, bundling is not performed.

5 Evaluation

To evaluate Ginnungagap, we have performed several experiments and present the vital results of our testing, with micro benchmarks of the RMI and migration functionality and live tests on PlanetLab (see (Erikstad, 2009) for further results and details).

5.1 Micro benchmarks

The micro benchmarks were performed using a local 100 Mbps network and ran on two machines with identical hardware (Intel Core 2 Duo E6750, 2.66GHz CPU, 2 GB of RAM) with Ubuntu 9.04 installed.

Remote Method Invocation: To test the RMI functionality and overhead, we invoked methods with several different combinations of return values and parameters. The results of these tests are shown in figure 6. Each method was invoked 10000 times. As expected, an overhead compared to a local invocation is introduced by an RMI, partially due to the overhead of (de)serializing parameters (not necessary for local invocations), and also the latency introduced by the network. We see a gradual increase in the time to complete RMI invocations of vectors of integers (as the vectors grow in size), but this is mainly due to the increased transmission time of the data. Apart from this, the mean difference in a remote and local invocation remains quite stable. We can also see that the standard deviation is relatively low. In summary, there is some overhead associated with an RMI compared to a local invocation, but this is to be expected. The primary obstacle is the latency introduced by the network, and the size of data being transmitted.

Migration: In the migration tests, we have performed two sets of tests, one where we migrate single objects of varying size, and one set where we vary the size of the object groups and their size. The results of migrating a single object are shown in figure 7. As expected, the time to complete a migration of a single object gradually increases with the size of the object.

The results of migrating groups of objects are shown in figure 8. Here we see much of the same pattern as migration of single objects, with the gradual increase in completion time being mainly affected by the size of the objects and the number

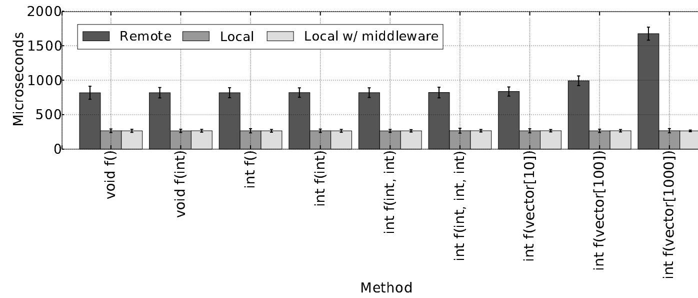


Figure 6 Average time of method invocation with stdev

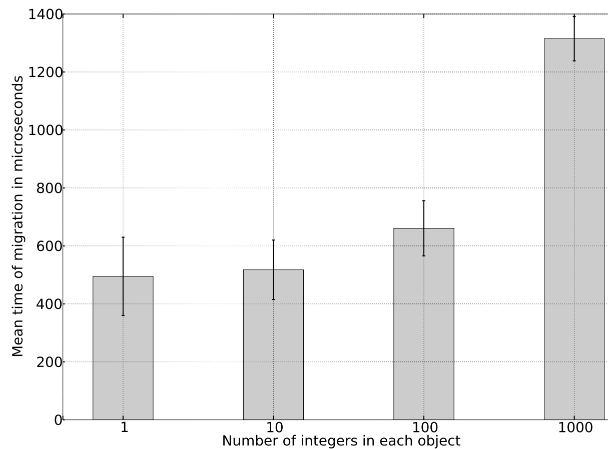


Figure 7 Average single object migration time with stdev

of groups. There is, however, more overhead introduced by the migration of groups, as the middleware has to perform additional work; it combines the objects into a single large message.

RMI during migration: We have also looked at the overhead introduced while performing RMI on an object while it is being migrated, and the results of this test are presented in figure 9. We continuously performed RMI to a group of objects that were being migrated back and forth between two servers at regular intervals. As we can see, it is clear that performing an RMI during migration adds overhead. This overhead is introduced because the name service has not been updated yet, and as such, RMIs are buffered until they can be forwarded. This buffering effect is witnessed in the greater variance displayed in the completion time for buffered/chained RMIs. Currently, we do not forward messages as they arrive, instead we wait for a `MIGACK` before forwarding the messages. It is reasonable to assume that this overhead could be reduced by changing this to an on-the-fly forwarding of the incoming messages. The messages would then be ready for processing as soon as the migration was finalized, effectively removing

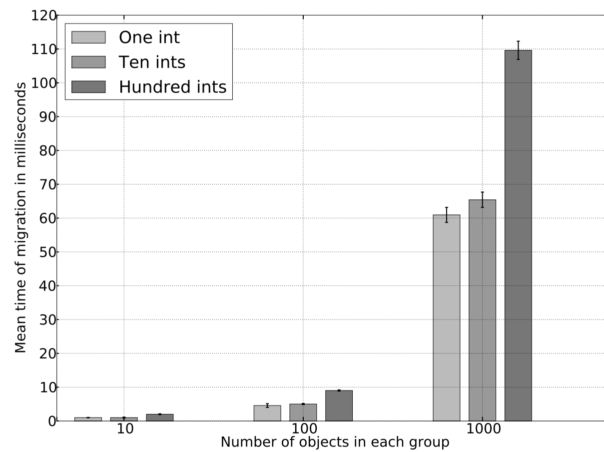


Figure 8 Average group migration time with stdev

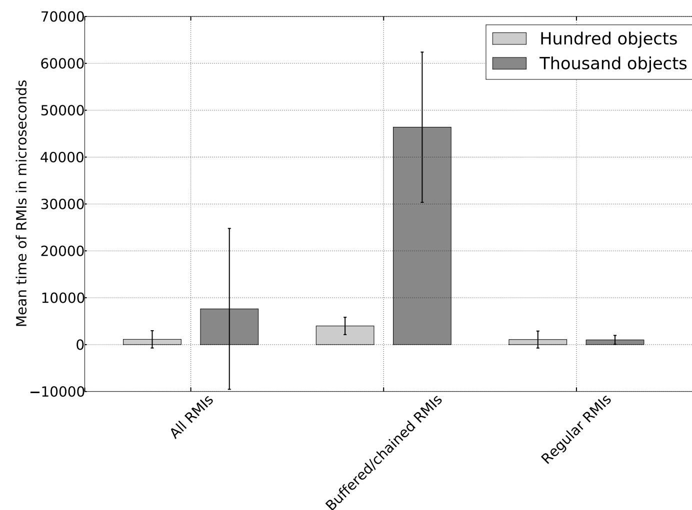


Figure 9 Average time of RMI during migration with stdev

some of the overhead (latency) introduced by having to wait for the forwarding of messages, thus reducing the variance in completion time for an RMI.

5.2 PlanetLab tests

We ran several different tests on the open world-wide research platform PlanetLab, primarily to test the usability of the core node selection algorithm and middleware itself. Also, the PlanetLab nodes that we selected as a server and proxies were those with little or no load in the PlanetLab network.

To test the core node selection algorithm, we ran a test where clients, approximately 120 of them, were initially connected to a main server. As more proxies were gradually added, the core node selection algorithm was activated (see figure 10 for an overview of the PlanetLab nodes included in the experiment).

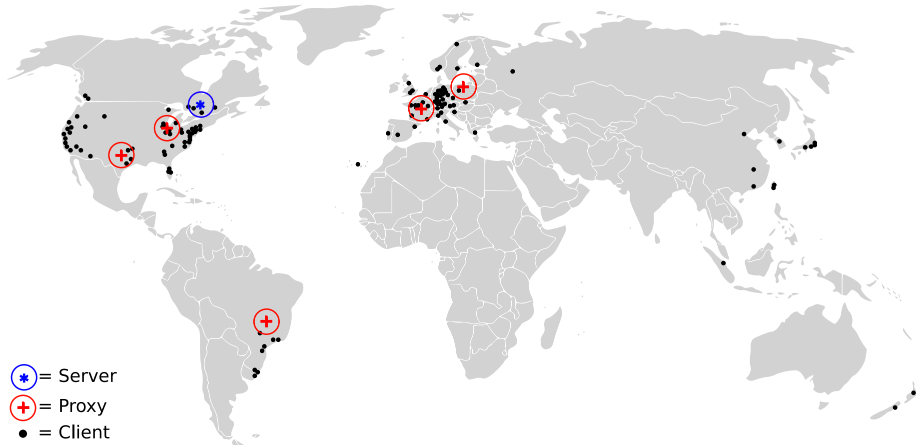


Figure 10 Geographical locations of PlanetLab nodes

The results of this test are summarized in figure 11 and 12. As we can see in figure 11, adding two additional proxies made it possible to reduce the aggregate mean latency of the interacting clients. Adding an additional three proxies further decreased the aggregate mean latency, but the gain was not as great. It is to be expected that the reduction does not necessarily improve greatly with a large number of added proxies, as there is always variance in the latencies of the connected clients, but it is clear that the gaming experience can be improved by the introduction of a small number of proxies. The crossover at the far left of figure 11 shows how a small number of connections benefit (when only a single server is available), while many suffer. By introducing a small number of proxies, performing a core-selection and migrating the state and interacting players to this proxy (if it is able to improve the aggregate latency of the players) the number of connections that benefit is reduced, but correspondingly the number of connections that suffer is also reduced.

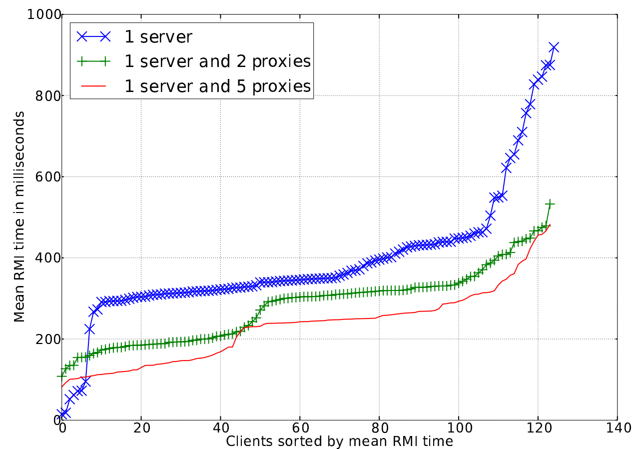


Figure 11 Aggregate mean of RMI

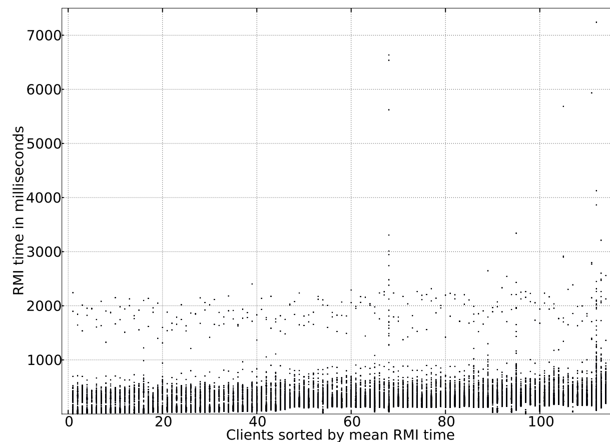


Figure 12 RMI time per client with varying numbers of proxies

In figure 12, we see the completion time of RMIs issued, as proxies are being added. We can see that most of the invocations are well below the 1000ms mark, but that there are at least a couple of invocations that take longer time. These are caused by migrations, as we noted during our micro benchmarks of RMI during migration, as seen in figure 9.

In figure 13, we see the results of a test simulating a small region in a virtual environment with players entering and exiting the region continuously. The setup consisted of 1 server, 5 proxies and approximately 120 clients. The session time of the clients follows an exponential distribution with an average scale of 60 seconds. The core-selection algorithm was activated every 30 seconds, with a migration occurring only if the minimum gain of each client was 2ms. This resulted in a total of 23 migrations, where all, except the first (from the server to a proxy) were between two of the proxies in the setup. The mean migration time was 813ms, excluding one exception, which was 28868ms, and we can see the resulting spikes in the RMI times of the sessions in figure 13. PlanetLab uses virtualization to schedule slices for run-time on the PlanetLab nodes. We believe this spike is a result of such virtualization, where the slice running our application was scheduled out. We did not see such spikes in our earlier PlanetLab tests, but these ran for shorter periods of time. This is one of the problems with running latency sensitive experiments in PlanetLab. An ICMP ping between the two proxies involved did not reveal any latency-related problems, but ICMP ping is not affected by the virtualization (because we use application layer ping, we are affected by virtualization). We ran the test several times with different configurations, but experienced similar spikes each time. We did not experience these in our micro benchmarks of the migration functionality. Apart from these spikes, most of the invocations during this test were well within the 1000ms mark. As such, it should be possible to perform migrations on regions with heavy traffic quite frequently, without adverse side-effects.

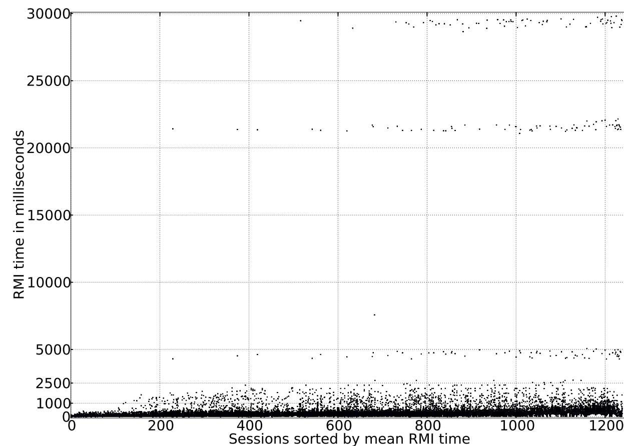


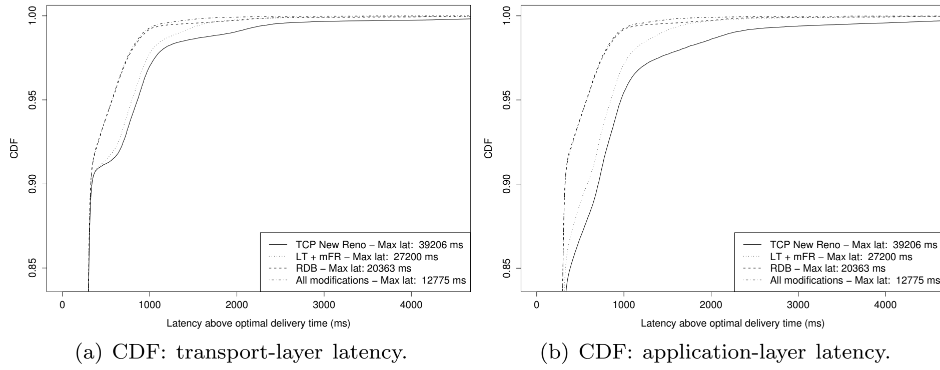
Figure 13 Mean RMI time of sessions

5.3 Latency effect of the TCP thin stream modifications

The modifications to TCP described in section 4 influence both the network monitoring, used for server selection, and the application data delivery. As our TCP modifications are implemented in the Linux kernel, we cannot use PlanetLab as a testbed (as we did in our previous experiments), because we are not permitted to modify the kernel on machines hosted in this network. As such, we had to use machines controlled by us instead.

5.3.1 Application data delivery

To observe the effects of latency at the application layer in a real-world Internet scenario, we replayed packet traces from Anarchy Online (approximately 1 hour of game play from Funcom's servers) between a machine located in Worcester, Massachusetts (USA) and a machine in Oslo (Norway). Both machines were connected to commercial access networks. In figure 14, the results are summarized. The observed path RTT was 116 ms, and the observed loss rate during this test was surprisingly high, 8.86% on average for the TCP New Reno stream. For the transport layer delivery latency (figure 14(a)), 9% of the traffic shows improved latency (0.91 on the y-axis), which is a reflection of the observed 8.86% loss rate, which comes as a result of the aggressive retransmission mechanisms. We also see that 99% of the data at the transport layer level is delivered within 1000 ms (the latency threshold where MMOG users start to get annoyed (Claypool et al., 2006)), while TCP New Reno delivers 97% within 1000 ms. On the application layer (figure 14(b)), where TCP's in-order delivery is accounted for, about 17% of the data is delivered faster using the modifications. With the modifications, about 99% was delivered within 1000 ms, while for TCP New Reno the corresponding percentage was 95%. Finally, the worst-case latencies were also greatly reduced, to about 1/3 of the New Reno results (39.2s versus 12.8s). The plots show that all mechanisms improve data delivery time when loss occurs, but RDB, being more



(a) CDF: transport-layer latency. (b) CDF: application-layer latency.

Figure 14 Anarchy Online packet trace replayed (Massachusetts - Oslo).

aggressive, is most efficient. Thus, our TCP modifications reduce the loss recovery experienced at the application level greatly, i.e., directly influencing the users' game experiences.

5.3.2 Server selection

Server selection is an important facet of GINNUNGAGAP. To make the best possible decision of whether to migrate game state (and the corresponding players) or not, it is important to have minimal variance in the gathered latencies. As such, we ran a test to determine the effect our TCP thin stream modifications would have on the variance of the gathered latencies, and subsequently the accuracy of the core selection algorithm. The results of this test are summarized in figure 15 (the source code of the test application is available for download at http://simula.no/research/networks/software/).

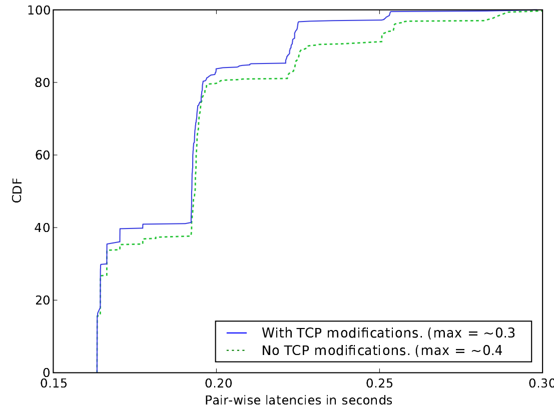


Figure 15 CDF of pair-wise latencies after core selection

For this test, we connected 20 clients to a server, where the clients measured their latencies using an application layer ping. For realistic results, we used *netem* to induce additional packet loss and latency on the client and server connections.

As reflected in section 5.3.1, we can experience peak loss rates of 9%. Thus, we used 1 and 4 percent (average) packet loss respectively for the server and client, and a latency of 70-120ms (randomized per connection and test). In our tests, the clients reported the average from 10 application layer measurements (see figure 15). The server then performed core selection using these measurements as input, and as such, the figure reflects the lowest pair-wise latency of the clients to the server (see section 3.1 for further details). The test ran in two iterations, one with TCP thin stream modifications enabled, and one where they were disabled.

From figure 15, we observe that the thin-stream modifications improve the decision of the core selection algorithm, because a greater percentage of the measurements are affected less by the presence of packet loss. This is so, because inaccuracies can only take the form of overestimated latency, which occurs when the retransmission mechanisms of unmodified TCP fail to recover lost packets quickly because packets rates are too low. The resilience to packet loss observed when the TCP modifications are enabled is thus a result of the aggressive retransmission strategies of the TCP thin stream modifications.

Basing a core selection decision on measurements gathered over time is preferential to single measurements, as multiple measurements converge towards the player's "real" latency. However, the scalability of such an approach is uncertain. One possibility is to have the server read the RTT value from TCP, based on in-game communication (state updates etc.), though in cases where no communication channel is open to (potential) proxies this problem has to be resolved in another way, such as active application layer probing.

6 Discussion

Due to the distributed name service resolving references to remote objects, migration can become a time-consuming task. An object that is frequently migrated creates trails of object references at the nodes it visits. We handle this scenario partially by embedding information about the emanating node in a message that is forwarded, so the reply can be sent directly to the calling node.

Another concern is related to the side-effects of migrating game-state. If migration is not performed transparently, we might adversely affect interacting players. In Nelson et al. (2005), it is demonstrated how an active application is moved from one virtual machine to another, with minimal perceived impact to the user's interaction with the application.

Partial failures occur when a node in a distributed system becomes unavailable, effectively rendering the objects managed by it inaccessible. The current framework does not accommodate for partial failures, but there are ways to minimize the repercussions of these incidents. One possibility is to distribute multiple copies of an object in the system, and utilize a peer-to-peer based look-up system such as Chord or Tapestry, to locate the object. However, the drawback of utilizing such an approach is the overhead introduced by having to keep the multiple instances of an object synchronized.

Core selection and migration require resources in the form of processing power and network messages when activated, as such, policies should govern their activation. Core-selection can be performed on the basis of churn (in the form

of players joining/leaving), time of day, player arrival rate, history (a preemptive approach, where known situations, such as battles, trigger the activation) and server-load. Migration can be performed on the basis of threshold of aggregate latency, number of players, packet loss, and server load. In addition to activation policies, it might also be useful to define cool-down periods for core-selection and migration, to avoid thrashing in the system.

To this effect, accurate measurements of the player latencies to the available servers and proxies are instrumental for making correct decisions. These measurements can, for instance, be retrieved from packets containing game state, but typically a client interacts with only one server at a time. And even in this case, the player may be vulnerable to the packet loss present in the Internet, and the inherent weaknesses of TCP as a transport protocol for thin streams. One possibility is to send a train of packets, and measure the mean of these. This, however, is costly, and the scalability of such a solution for a large number of players, such as we commonly witness in MMOGs, is uncertain. As such, there is much to be gained by utilizing thin stream modifications to TCP, which are able to accommodate for the packet loss in the Internet for thin streams.

Core selection depends on full knowledge of the network, which we can obtain by observing TCP latency. Latency estimation techniques provide an alternative, and in (Beskow et al., 2009b; Vik et al., 2009), we examined the feasibility of using estimated latencies for situations when real-time measurements are unable to scale to the number of interacting players. These active probing techniques generate an amount of traffic that has roughly the same intensity as a modern game's traffic itself. Since the TCP thinstream modifications minimize the variance in the RTT measurements available from the TCP connection structs we have considered using such a passive probing approach, based on the TCP connections.

The proposed framework is tested in a client-server setting, but in general, the framework should also be useful in a peer-to-peer scenario trying to dynamically select a server among the peers. Thus, as the peers join and leave the game, the selection of peers controlling the game could be performed using our middleware.

The implemented thin stream TCP modifications may increase the number of spurious transmissions (Petlund, 2009). However, the reduction in latency, both for application data delivery and latency monitoring for core selection, justifies the need to suppress occasional spurious retransmissions.

7 Conclusion

High latency may be devastating for the game play experience. An important factor for world-spanning games, such as MMOGs, lies in the diversity of its user base where there will, at any point in time, be a number of players connected from different physical locations. The large challenge is that the group of players in the game or in the game region is dynamic, and using a statically selected server may therefore give good results in one moment but poor in the next.

Beskow et al. (2009a) describe the implementation of this middleware and presented experimental results showing the performance gains from in-house lab experiments and running a simple game on PlanetLab. Here, we have expanded on these ideas and tested the effects of our TCP thin stream modifications

for improving the resilience of latency measurements and for improving the throughput of game state in the network. As a result, players receive data sooner, and the core selection algorithm is able to make a better decision with regard to the optimal placement of game state.

As future work, we would like to review the possibility of optimizing the placement for smaller, highly coupled groups of interacting players within a region. This can be achieved by applying graph partitioning algorithms or similar techniques for isolating these groups.

Bibliography

ARMITAGE, G. (2008a) Client-Side adaptive search optimisation for online game server discovery. In *Lecture notes in computer science 4982*, pp. 494-505.

ARMITAGE, G. (2008b) Optimising online fps game server discovery through clustering servers by origin autonomous system. In *Proceedings of International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'08), Braunschweig, Germany*, pp. 3-8.

BESKOW, P. (2007) Migration of objects in a middleware for distributed real-time interatctive applications. Master's thesis, Department of Informatics, University of Oslo, Norway.

BESKOW, P., VIK, K.-H., GRIWODZ, C., AND HALVORSEN, P. (2008) Latency reduction by dynamic core selection and partial migration of game state. In *Proceedings of NetGames'08, Worcester, MA, USA*, pp. 79-84.

BESKOW, P., ERIKSTAD, G., GRIWODZ, C., AND HALVORSEN, P. (2009a) Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Proceedings of NetGames'09, Paris, France*, pp. 1-6.

BESKOW, P., VIK, K.-H., HALVORSEN, P., AND GRIWODZ, C. (2009b) The partial migration of game state and dynamic server selection to reduce latency. In *Springer's Multimedia Tools and Applications Special Issue on Massively Multiuser Online Gaming Systems and Applications 45*, 1-3, pp. 83-107.

BRUN, J., SAFAEI, F., AND BOUSTEAD, P. (2006) Server topology considerations in online games. In *Proceedings of NetGames'06, New York, NY, USA*, p. 26.

CHAMBERS, C., CHANG FENG, W., CHI FENG, W., AND SAHA, D. (2003) A geographic redirection service for on-line games. In *Proceedings of ACM International Conference on Multimedia, Berkeley, CA, USA*, pp. 227-230.

CHAMBERS, C., CHANG FENG, W., SAHU, S., AND SAHA, D. (2005) Measurement-based characterization of a collection of on-line games. In *Proceedings of ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA*, pp. 1-14.

CHANG FENG, W., CHANG, F., CHI FENG, W., AND WALPOLE, J. (2002) Provisioning on-line games: a traffic analysis of a busy Counter-strike server. In *Proceedings of ACM SIGCOMM Workshop on Internet measurement, Marseille, France*, pp. 151-156.

CHANG FENG, W., AND CHI FENG, W. (2003) On the geographic distribution of on-line game servers and players. In *Proceedings of NetGames'03, Redwood City, CA, USA*, pp. 173-179.

CLAYPOOL, M. (2005) The effect of latency on user performance in real-time strategy games. In *Elsevier Computer Networks 49*, 1, pp. 52-70.

CLAYPOOL, M. (2008) Network characteristics for server selection in online games. In *Proceedings of Multimedia Computing and Networking (MMCN'08), San Jose, CA, USA*, pp. 681808.

CLAYPOOL, M., AND CLAYPOOL, K. (2006) Latency and player actions in online games. In *Communications of the ACM 49*, 11, pp. 40-45.

DICK, M., WELLNITZ, O., AND WOLF, L. (2005) Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of NetGames'05, Hawthorne, NY, USA*, pp. 1-7.

ERIKSTAD, G. A. (2009) Latency reduction in distributed interactive applications by core node selection and migration. Master's thesis, Department of Informatics, University of Oslo, Norway.

LEE, K., KO, B., AND CALO, S. (2005) Adaptive server selection for large scale interactive online games. In *Computer Networks 49*, 1, pp. 84-102.

NELSON, M., LIM, B., AND HUTCHINS, G. (2005) Fast transparent migration for virtual machines. In *Proceedings of USENIX Annual Technical Conference*, pp. 25-25.

PETLUND, A. (2009) *Improving latency for interactive, thin-stream applications over reliable transport*. PhD thesis, Department of Informatics, University of Oslo, Norway.

PETLUND, A., EVENSEN, K., GRIWODZ, C., AND HALVORSEN, P. (2008) Improving application layer latency for reliable thin-stream game traffic. In *Proceedings of NetGames'08, Worcester, MA, USA*, pp. 91-96.

VIK, K.-H. (2009) *Group Communication Techniques in Overlay Networks*. PhD thesis, Department of Informatics, University of Oslo, Norway.

VIK, K.-H., GRIWODZ, C., AND HALVORSEN, P. (2009) On the influence of latency estimation on dynamic group communication using overlays. In *Proceeding of Multimedia Computing and Networking (MMCN'09), San Jose, CA, USA*, 7253, pp. 725307.

Paper VI

Kahn process networks are a flexible alternative to MapReduce

Zeljko Vrba, Paul B. Beskow, Pål Halvorsenand Carsten Griwodz

Published: Proceedings of 11th IEEE International Conference on High Performance Computing and Communications (HPCC), ed. by NA, pp. 154-162, IEEE Computer Society, 2009

Evaluation: 249 papers submitted, 54 accepted, with acceptance rate of 21.6%

Author contribution: Vrba was the driving force behind this article and was responsible for the implementation of the Nornir run-time. Beskow contributed with knowledge on processing systems and investigations into related work.

Kahn process networks are a flexible alternative to MapReduce

Željko Vrba, Paul Beskow, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory and University of Oslo, Norway
{zvrba,paulbb,paalh,griff}@ifi.uio.no

Abstract

Experience has shown that development using shared-memory concurrency, the prevalent parallel programming paradigm today, is hard and synchronization primitives nonintuitive because they are low-level and inherently non-deterministic. To help developers, we propose Kahn process networks, which are based on message-passing and shared-nothing model, as a simple and flexible tool for modeling parallel applications. We argue that they are more flexbile than MapReduce, which is widely recognized for its efficiency and simplicity. Nevertheless, Kahn process networks are equally intuitive to use, and, indeed, MapReduce is implementable as a Kahn process network. Our presented benchmarks (word count and k-means) show that a Kahn process network framework permits alternative implementations that bring significant performance advantages: the two programs run by a factor of up to ~ 2.8 *(word-count) and* ~ 1.8 *(k-means) faster than their implementations for Phoenix, which is a MapReduce framework specifically optimized for executing on multicore machines.*

1 Introduction

Developing applications that exploit multiple computing resources, be it multiprocessors, chip-level multiprocessors or co-processors is challenging. Since such systems have become a commodity, the number of developers expected to face this challenge is increasing. Implicit communication, through shared data structures, is the reigning paradigm despite the recognition that standard concurrency control mechanisms (e.g., mutexes, semaphores and condition variables) are difficult to use correctly [12]. Their non-deterministic nature also makes the learning curve steeper – for example, many newcomers to multithreaded programming are surprised to learn that mutexes or condition variables wake up threads in arbitrary instead of FIFO order. Hence, several frameworks and higher-level abstractions (designed to ease parallelization) have been proposed, such as software transactional memory (STM) [9], MapReduce [6] and Dryad [10].

Each of these proposed frameworks, however, has some drawbacks: STM has large overheads; MapReduce is very rigid (in the sense that computation steps are predetermined); Dryad supports non-deterministic constructs and disallows cycles in the communication graph and is thus, like MapReduce, unable to model iterative algorithms.

We therefore propose that Kahn process networks (KPN) [11] be used for expressing parallelism in programs. KPNs are *the least restrictive message-passing model that yields provably deterministic programs*, i.e., programs that yield always the same output given the same input, regardless of the order in which individual processes are scheduled. Determinism is achieved by placing restrictions on processes; it has been proven formally that the resulting model is no longer deterministic if one or more of these restrictions are removed. Using KPNs for development of parallel applications brings several benefits:

- *Sequential coding of individual processes.* Processes are written in the usual sequential manner; synchronization is implicit in explicitly coded communication primitives (message send and receive).
- *Composability.* Connecting the output of a network computing function $f(x)$ to the input of a network computing $g(x)$ guarantees that the result will be $g(f(x))$. Thus, components can be developed and tested individually, and later assembled together to achieve more complex tasks.
- *Reliable reproduction of faults.* Because of determinism, it is possible to *reliably* reproduce faults (otherwise notoriously difficult), which will greatly ease debugging.

While MapReduce and Dryad also have most of the above benefits, KPNs have several additional key properties that make them suitable for modeling and implementing a wider range of problems than MapReduce and Dryad:

- *Arbitrary communication graphs.* Whereas MapReduce and Dryad restrict developers to the structure of figure 1 and directed acyclic graphs (DAGs), respectively, KPNs allow *cycles* in the graphs. Because of

this, they can directly model iterative algorithms. With MapReduce and Dryad this is only possible by manual iteration, which incurs high setup costs before each iteration [13].

- *No prescribed programming model.* Unlike MapReduce, KPNs do not require that the problem be modeled in terms of processing over key-value pairs. Consequently, transforming a sequential algorithm into a Kahn process often requires minimal modifications to the code, consisting mostly of inserting communication statements at appropriate places.

The flexibility of KPNs allows the implementation of the full semantics of MapReduce, so they are as easy (or hard) to use as MapReduce. However, we and others [4] have noticed that the structure of the MapReduce computation is not always a good match to the task at hand, and, as we will show in this paper, can adversely impact performance. Because a single, multi-core machine has much more limited resources than a large cluster, we deem that it is important to investigate MapReduce alternatives that will be more efficient on a small scale.

To investigate whether direct KPN modeling has advantages over modeling with MapReduce, we have implemented the word count, which is the canonical MapReduce example, and k-means applications. Each application is implemented in two variants, KPN-MR and KPN-FLEX (4 programs in total). The KPN-MR variant constructs a MapReduce network topology, while the KPN-FLEX variant constructs a topology tailored to the problem. In our benchmarks, KPN-FLEX outperforms KPN-MR by a factor of up to 1.3 for the k-means program, and by a factor of up to 6.7 in the word count program. Similarly, KPN-FLEX runs faster by a factor of up to ~ 2.8 (word-count) and ~ 1.8 (k-means) than their equivalents for Phoenix [13], which is a MapReduce framework specifically optimized for executing on multicore machines.

2 Related work

MapReduce [6] is a popular framework for processing of large amounts of data on a cluster of machines. Its basic structure is a pipeline, but each stage consists of several concurrent processes (see figure 1). The splitter process (S) splits the input into roughly equal chunks and sends them to m processes in the map stage. Processes in the map stage apply a user-defined function to each record, consisting of a key and a value,[1] and send the resulting records to n processes in the reduce stage. Records with identical keys are always sent to the same reduce process, so that the final value of the reduce function for the given key can be computed without communicating with other reduce processes.

[1]The key-value paradigm stems from the MapReduce pattern; it is not necessary to use it when modeling a problem as a KPN.

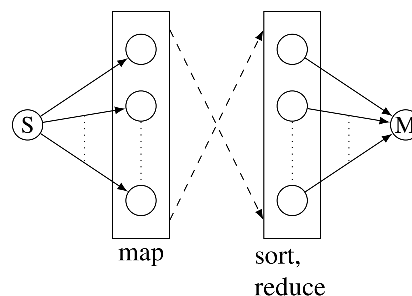


Figure 1. Structure of MapReduce computation represented as a KPN. Dahsed lines represent connections between all pairs of processes in the two stages. Merge (M) and split (S) processes can be omitted when several MapReduce computations are chained.

The reduce stage first sorts the data items by their key, and then applies another user-defined function over each group of data items with identical keys. The final stage of the computation is the merge process (M) which merges n sorted outputs from reduce processes into a single sorted output stream of data items.

Dryad [10] is a system for describing and executing, in a potentially distributed manner, computations whose communication patterns are expressed by directed *acyclic* graphs. Main features, namely message-passing and sequential programming model of individual processes, are shared with the KPN model. However, there are a number of differences. First, Dryad is not based on any formal foundation. Second, Dryad introduces asynchronous interfaces which may, as a consequence, result in nondeterministic programs (indeed, Dryad's nondeterministic merge is built upon these interfaces). Third, loops in the communication graph are not allowed, which makes it impossible to model iterative algorithms such as k-means (see section 5).

While there are other parallel programming models, due to space constraints we have been able to describe only MapReduce and Dryad in detail. Other models cover shared-memory models, pipelines, and unstructured point-to-point communication, as available through the MPI programming interface. In this taxonomy, while (parallel) pipelines and DAGs have limited expressiveness, i.e., do not allow cycles in the graph. This is possible with unstructured communication, but inadvertent problems – such as non-determinism and deadlocks – occur easily. In this taxonomy, KPNs are the most flexible model which still guarantees determinism, which is the reason for our deeper investigation of their properties.

3 Kahn process networks

A KPN [11] has a simple representation in the form of a *directed graph* with *processes* as nodes and *communication*

channels as edges (see Section 5 for examples). A process encapsulates data and a single, sequential control flow, independent of any other process. Processes are not allowed to share data and may communicate only by sending messages over channels. Channels are *infinite* FIFO queues that store discrete *messages*. Channels have *exactly one* sender and one receiver process on each end (1:1), and every process can have multiple *input* and *output* channels. Sending a message to the channel always succeeds, but trying to receive a message from an empty channel *blocks* the process until a message becomes available. It is not allowed to poll a channel for presence of data. These properties fully define the *operational semantics* of KPNs and make the Kahn model *deterministic*, i.e., the history of messages produced on the channels does not depend on the order in which the processes are executed, provided the scheduling is *fair*, i.e., that execution of a ready process will not be indefinitely postponed.

The theoretical model of KPNs described so far is idealized in two ways: 1) it places few constraints on process behavior, and 2) it assumes that channels have infinite capacities. These assumptions are somewhat problematic because they allow construction of KPNs which need unbounded space for their execution, but any real implementation is constrained to run in finite memory. A common (partial) solution to this is to assign *capacities* to channels and redefine the semantics of send to *block* the sending process if the delivery would cause the channel to exceed its capacity. Under such send semantics, an *artificial deadlock* may occur, i.e., a situation where a cyclically dependent subset of processes blocks on send, but which would continue running in the theoretical model. The algorithm of Geilen and Basten [7] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarging it by one message, thus resolving the deadlock.

4 KPN implementation

Our KPN execution environment is implemented in C++, and it runs on Windows and POSIX operating systems (Solaris, Linux, etc.) Our implementation[2] of the run-time environment for executing KPNs consists of a Kahn process (KP) scheduler, message transport and deadlock detection and resolution algorithms.

To the best of our knowledge, there exist only two other general-purpose KPN runtime implementations: YAPI [5] and Ptolemy II [3]. YAPI is not a pure KPN implementation, as it extends the semantics and thus introduces the possibility of non-determinism, its code-base is too large for easy experimentation (120 kB vs. 40 kB in our implementation), and we were unable to make it use multiple CPUs.

[2] Available code: `http://simula.no/research/networks/software`

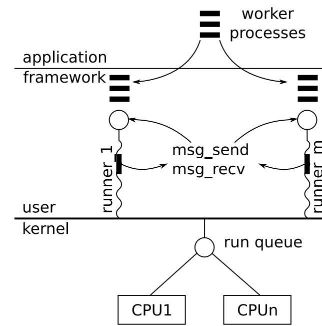


Figure 2. Two-level KPN scheduling

Ptolemy II is a Java-based prototyping platform for experimenting with various models of computation, and it spawns one thread for each Kahn process, which is rather inefficient for large networks. The amount of code that the JVM consists of would make it prohibitively difficult to experiment with low-level mechanisms, such as context-switch.

4.1 Process scheduler

The scheduler may be configured, at compile-time, in two ways. In the first configuration, each KP is run in its own OS-thread. Channels are protected by blocking mutexes, and notifications are done by using condition variables. However, our earlier measurements have shown that the native mechanisms suffer from high overheads, so we have also implemented an optimized KP scheduler.

The second configuration uses our own work-stealing scheduler. When the KPN is started, m runner threads are created and scheduled by the OS onto n available CPUs (see figure 2). Each runner implements a work-stealing policy [2], i.e., it has a private run queue of ready KPs, and if this queue is empty, it tries to steal a KP from a randomly chosen runner. For simplicity, we do not use the non-blocking queue described in [2]; instead we use ordinary mutexes. Context-switch between KPs is implemented in user-mode. On Solaris and Linux running on AMD64 architecture we employ hand-crafted assembly code; on other platforms we use OS-provided facilities, which often incur some additional overhead.

The channels are protected with *polling* mutexes: if a KP cannot obtain the channel's lock, it will spin, explicitly yielding to the scheduler between iterations, until it has finally obtained the lock. Waiting and signaling are implemented by a protocol between KPs and the scheduler. The protocol is optimized for the case of having at most one sleeping KP, which is possible because channels are 1:1, and at most one process can ever be blocked on a channel.

4.2 Message transport

Channels have a two-fold role: to transport messages and to interact with the scheduler, i.e., block and unblock pro-

cesses on either side of the channel. Message send/receive is implemented by copying the messages to/from channel buffers. Zero-copy transfer would require dynamic memory allocation, and we have measured that copying is less expensive as long as messages are smaller than ~ 256 bytes.

We have also extended channels with *EOF indication*: the sender can set the EOF status on the channel when it has no more messages to send. After EOF on the channel has been set, the receiver is able to read the remaining buffered messages, but the next receive will immediately return false instaed of blocking. A further attempt to receive a message from the channel will permanently block the process.

An alternative approach to EOF signaling is sending a message with specific contents as the last message on the channel. The disadvantage is that all values of the channel's type (e.g., `int`) might be meaningful in a given context, so no value could be used to encode the EOF value. In such cases, one would be forced to use more cumbersome solutions that also potentially impose additional overhead.

4.3 Deadlock detection and resolution

Deadlock detection and resolution is a mechanism which allows execution of KPNs in finite space. Since communication is 1:1, every cycle of blocked KPs is a ring; a property which greatly simplifies detection. Whenever the currently running KP would block on send, the algorithm is invoked to check whether an artificial deadlock occurred. If no cycle is found, the current KP is blocked and this fact is recorded in the blocking graph data structure. Otherwise, the capacity of the smallest channel in the cycle is increased by one, as suggested by [7]. If the channel belongs to the current KP, the current KP is immediately resumed; else the KP on the channel's send side is unblocked, and the current KP is blocked. Similarly, receiving from a full channel unblocks the KP on the sending side and removes an edge from the blocking graph.

Our current implementation uses a centralized data structure for the blocking graph, so the above operations must run while holding a single global mutex. Despite this, we have not noticed significant scalability issues on up to 8 CPUs on the workloads described in this paper. This is because of the interaction of the following elements:

- KPs that do not need to block or wake up another KP continue to run undisturbed.
- Since the number of KPs is usually much larger than the number of runners, it is to be expected that many KPs will be ready and little work will be stalled.
- The time the global mutex is held for in the worst case is proportional to the size of the largest potential cycle, which is small in our examples.

5 Modeling with KPNs

Parallelizing an application with KPNs (as well as with Dryad and MapReduce) entails three steps:

1. Identifying independent subtasks (components) and their corresponding inputs and outputs.
2. Implementing subtasks, possibly by "filling in the blanks" in off-the-shelf components (e.g., Reduce and n-way merge).
3. Determining the number of subtasks and communication between them; the latter will be largely dependendent on the previous step.

We demonstrate this methodology on the word count and k-means programs. Even though word count is the "canonical" MapReduce example, its implementation via MapReduce is rather inefficient, in terms of the amount of unnecessary extra work done. k-means is an example of an *iterative* data-parallel algorithm, and is because of that a rather bad match for MapReduce, as is also noted in [13].

5.1 Word count

The word count program counts the number of occurrences of each word in a given text and outputs them sorted in the order of decreasing frequency.

The MapReduce and KPN-MR solutions require *two* MapReduce instantiations connected in series, such that the output of the Reduce stage is sent directly to the input of the Map stage of the second instantiation, without an intervening merge stage. Schematically, the network looks like this (note the pipeline structure):

$$\mathrm{S} \to \mathrm{MR}_1 \to \mathrm{MR}_2 \to \mathrm{M}$$

where each MapReduce has a structure corresponding to the one shown in figure 1.

The output of MR_1 is a list of word-count pairs sorted alphabetically by word. MR_2 then reverses word-count pairs to count-word pairs, count now being the key, in order to sort them by frequency. The reduce function of MR_2 is identity and the only role of this stage is sorting.

The subtasks of our KPN-FLEX solution (see Figure 4) are operationally identical to those of the KPN-MR solution: splitting the (memory-mapped, copy-on-write) input file into chunks, counting word frequency in individual chunks, summing and sorting partial word frequencies of chunks, and finally merging partial counts into a single sorted list.

The code that implements the count stage is shown in Figure 3; note that this is *real code*, taken from a working program (see appendix of [6] for Google's implementation). Kahn processes are in our framework implemented

```
void count::parse_word(text_chunk &chunk)
{
  char *b = get<0>(chunk), *e = get<1>(chunk);
  char *w;

  // Skip leading non-letters.
  while((b < e) && !inword(*b)) ++b;


  // Parse word and convert to uppercase
  for(w = b; (w < e) && inword(*w); ++w)
    *w -= ((*w >= 'a') && (*w <= 'z'))*('a'-'A');

  // Insert new word, or increase count
  if(b != w) {
    std::pair<count_hash::iterator,bool> rv =
      counts_.insert(count_hash::value_type(
        word(b, w), 1));
    if(!rv.second)
      ++rv.first->second;
  }
  get<0>(chunk) = w;
}

void count::behavior()
{
  text_chunk chunk;
  boost::hash<word> h;
  count_hash::iterator it;

  in(0).recv(chunk);
  while(get<0>(chunk) != get<1>(chunk))
    parse_word(chunk);
  for(it = counts_.begin();
    it != counts_.end(); ++it)
      out(h(it->first) % out_count()).send(it);
  eof_all();
}
```

Figure 3. C++ code for the count stage of the word count program in KPN-FLEX implementation (see also figure 4). The code in slanted font (lines 31 and 34–37) are the only necessary additions to the sequential version of the algorithm.

as classes which must derive from the `actor` class and implement the pure virtual `behavior` method, which is the entry point.

The `count` process initially receives a text chunk which is represented as a (start, end) tuple of pointers. The following `while` counts individual words in the text chunk: it keeps calling the `parse_word` routine which extracts the next word from the chunk and updates the hash table of counts. When all words from the chunk have been extracted, the hash table is iterated and iterators to each (word, count) pair are sent to the sum process. To distribute the load approximately evenly, the destination output is calculated by taking the hash of each word modulo the number of outputs. Finally, the `eof_all()` statement sets an EOF indication on all output channels.

5.2 k-means

k-means is an iterative algorithm used for partitioning a given set of points in multidimensional space into k groups; it is used in data mining and pattern recognition. The algorithm consists of the following steps:

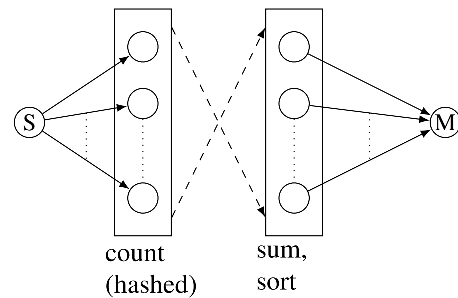


Figure 4. KPN-FLEX for solving the word count problem.

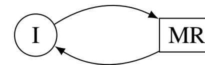


Figure 5. KPN-MR for solving the k-means problem. We had to introduce additional process (I) to iterate the MR block, whose constituent parts are shown in Figure 1.

1. Make initial guess for the center of gravity (centroid) of each of the k groups.
2. Assign each point from the dataset to the group which centroid is closest to the point.
3. Recalculate new centroids based on the new assignment of points to clusters.
4. Repeat from step 2, now with new centroid coordinates, until the process converges (i.e., the centroids do not move by more than a preset threshold).

In the KPN-MR implementation, the Map stage computes the index i of the nearest mean for each point p and emits (i, p) as the intermediate key-value pair. The Reduce stage computes the new mean values from the itermediate key-value pairs. Since MapReduce alone cannot model iterative algorithms, we construct the KPN shown in Figure 5. The MapReduce implementation is optimized in the same way as for word count: the Map stage and Reduce stage send out pointers to vectors instead of individual points.

KPN-FLEX that executes the k-means algorithm is shown in Figure 7. The I (iterate) process has two phases: initialization and iteration. The initialization phase generates a vector of random points, selects new k random points as the starting centroids, and partitions the point vector into as many approximately equal parts as there are worker processes (unlabeled in the figure). Then, it sends a partition of the point set to each worker as a pair of (start, end) iterators, and sends all k centroids, each in own message, to every worker. When the initialization phase has finished, the iteration phase, described below, begins.

A worker receives on its input the current location of the means and computes the new cluster assignment for its part of the point set; each time a point is assigned to a new cluster, the centroid corresponding to the partial point set assigned to the worker is updated. When a worker has finished with its points, it sends partial sums for centroids to the iteration process (I). Since mean is a linear operation, the iteration process can take the partial sums and counts to compute the new locations of the centroids. If the new locations are equal to the locations from the previous iteration, the process has converged and the program terminates.

The full code executed by the worker process is shown in Figure 6. The additional code necessary to transform the sequential algorithm into a Kahn process is shown in slanted text. Also note how the code in slanted font resembles programming with *ordinary files*: a process reads its input, item by item, from input ports (files opened for reading), does some processing, writes results to its output ports (files opened for writing), and exits when EOF is detected on some input port (file).

```
1  void kmeans::behavior()
2  {
3    size_t i;
4
5    // Receive the point set.
6    in1.recv(points_);
7
8    while(1) {
9      point_vec partial_sums(cur_means_.size(),
10       ZEROPOINT);
11
12     // New iteration: get recalculated means
13     for(i = 0; i < cur_means_.size(); ++i)
14       if(!in2.recv(cur_means_[i]))
15         return;
16
17     point_vec::iterator it;
18     for(it = get<0>(points_);
19       it != get<1>(points_); ++it)
20     {
21       // Recompute point's cluster.
22       int c = std::min_element(
23           cur_means_.begin(), cur_means_.end(),
24           bind(&distance, *it, _1) <
25           bind(&distance, *it, _2))
26         - cur_means_.begin();
27
28       // Assign point to the new cluster and
29       // update cluster's partial sum
30       it->c = c;
31       for(int k = 0; k < DIM; ++k)
32         partial_sums[c].v[k] += it->v[k];
33       ++partial_sums[c].c;
34     }
35
36     // Send partial sums to the I process.
37     for(i = 0; i < partial_sums.size(); ++i)
38       out.send(partial_sums[i]);
39   }
40 }
```

Figure 6. C++ code for the worker processes of the k-means program in KPN-FLEX implementation. The code lines typeset in slanted font (lines 13–15, 37–38) are the only necessary additions to the sequential algorithm.

6 Evaluation

To evaluate advantages of modeling with KPNs, we have implemented the word count and k-means programs as KPN-FLEX and KPN-MR networks (see Section 5). All test programs have been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). We have run them on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM running linux kernel 2.6.27.3. Each program has been run 10 times on 1, 2, 4 and 8 CPUs with differing number of worker processes. We present the average *wall-clock (real) time* of 10 runs together with error lines showing the standard deviation. Note that we compare the *unoptimized* (in terms of the number of exchanged messages) KPN implementations with the *optimized* MapReduce (over KPN) implementations.

The wall-clock time metric is most representative because it accurately reflects the real time needed for task completion, which is what the end-users are most interested in. We have also measured system and user times (`getrusage`), but do not use them to present our results because 1) they do not reflect the reduced running time with multiple CPUs, and 2) resource usage does not take into account sleep time, which nevertheless may have significant impact on the task completion time.

6.1 Implementation considerations

Our *unoptimized* MapReduce implementation of the word count program was slower by an order of magnitude than the optimized implementation, for which we present the results. In the unoptimized case, each key-value pair was sent in its own message, and the number of sent messages was dominated by the *total* number of words in the input file. The *optimized* KPN-MR implementation allocates as many vectors holding key-value pairs as there are outputs from the Map and Reduce stages. Individual pairs are distributed across the vectors, taking care that pairs with the same key are placed in the same vector. Afterwards, a *single* message with a pointer to the vector is sent to each output. We could have optimized the KPN-FLEX implementations in a similar way, but we decided against it because it would somewhat obscure similarity with the sequential algorithm.

We have furthermore optimized the KPN-MR k-means implementation by introducing the extra process that iterates the MapReduce block (see Figure 5), thus avoiding KPN startup and shutdown costs which Phoenix suffers in each iteration.

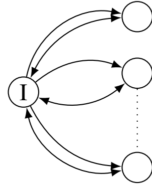


Figure 7. KPN-FLEX for solving the k-means problem. Edges with double arrows represent two channels, one in each direction. "Parallel" channels are carrying messages of different types.

6.2 Word count

We have run the word count program on files of size 10, 50 and 100MB, which is also the dataset used to benchmark Phoenix. Both implementations have been run on 1, 2, 4 and 8 CPUs, and with the number of runners in each stage varying from 8 to 128 in steps of 8. Figure 8 shows running time of the KPN-FLEX solution for the three datasets in the left column, and of KPN-MR solution in the right column; our findings are also summarized in Table 1.

We can see that the running time decreases as more CPUs are used and that the KPN-FLEX version has several *times* better performance than the KPN-MR version. The variation in running times is rather small between experiments, except for the KPN-MR solution with 2 runner threads; as of now we cannot explain this variation. With respect to the number of workers, the situation is more complicated: the running time *decreases* as the number of workers *increases* up to a certain point. Word count is a communication-intensive task and having more workers improves performance by more evenly distributing the load among CPUs and reducing contention in the work stealing scheduler: the more processes there are in the system, the smaller probability that a runner will need to steal a process from another runner.

Furthermore, as described in Section 4, processes acquire locks by busy-waiting and yielding between successive attempts. When there are enough ready processes, waiting on a lock will yield to another ready process which will be able to perform some useful work. When the number of processes increases even more, the performance starts dropping again for two reasons: 1) context-switch overheads (which also increase pressure on CPU caches), and 2) we conjecture that an even more important factor is contention over the single deadlock detection lock.

Table 1 gives an insight into scalability of KPN-MR and KPN-FLEX implementations with respect to the number of CPUs, which turns out to be approximately *logarithmic*: the running time decreases *linearly* with each *doubling* of the number of runner threads. We see also that KPN-FLEX is consistently faster than KPN-MR, by a factor of 3 – 6.7, and that the speedup is *proportional with the problem size*.

Size	KPN-MR			KPN-FLEX			
	n	t	α	n	t	α	f
10/1	96	1.24	1.00	8	0.36	1.00	3.41
10/2	64	0.80	1.55	16	0.21	1.71	3.80
10/4	48	0.46	2.70	16	0.14	2.57	3.24
10/8	32	0.32	3.88	16	0.11	3.27	2.95
50/1	120	8.57	1.00	8	1.54	1.00	5.57
50/2	128	5.14	1.67	16	0.87	1.77	5.91
50/4	64	2.92	2.93	24	0.52	2.96	5.57
50/8	48	1.86	4.61	24	0.37	4.16	5.06
100/1	128	20.00	1.00	8	3.17	1.00	6.30
100/2	88	12.22	1.64	32	1.81	1.75	6.74
100/4	104	6.65	3.01	32	1.08	2.94	6.14
100/8	56	4.23	4.73	32	0.74	4.28	5.69

Table 1. Word count experiment summary: number of workers n that achieves the best running time t (in seconds), relative speedup over one CPU (α) and speedup factor of KPN-FLEX over KPN-MR (f).

	Points	Groups	Iterations
A	100000	100	97
B	200000	50	140
C	200000	100	139

Table 2. k-means problem sizes and number of iterations until algorithm converges. The amount of work performed in each iteration is proportional to the product of the number of points and groups.

6.3 k-means

The k-means program generates a number of 3-dimensional points with random integer coordinates from the cube $[0, 1000)^3$. The random number generator is always initialized with the same seed, so each run executes the same number of iterations. We have measured performance of the program on three problem instances differing in the number of points and groups (see Table 2) and on 1, 2, 4, and 8 CPUs. Since this is a CPU-intensive benchmark with little communication, we have set the number of workers to be equal to the number of runner threads; very little experimentation was needed to establish that this was the optimal choice.

Again, the KPN-FLEX solution outperforms the KPN-MR solution, although the speedup is not as drastic as in the word count example. The main reason for this is that, unlike with word count, very little unnecessary work is performed by KPN-MR, namely only sorting before comput-

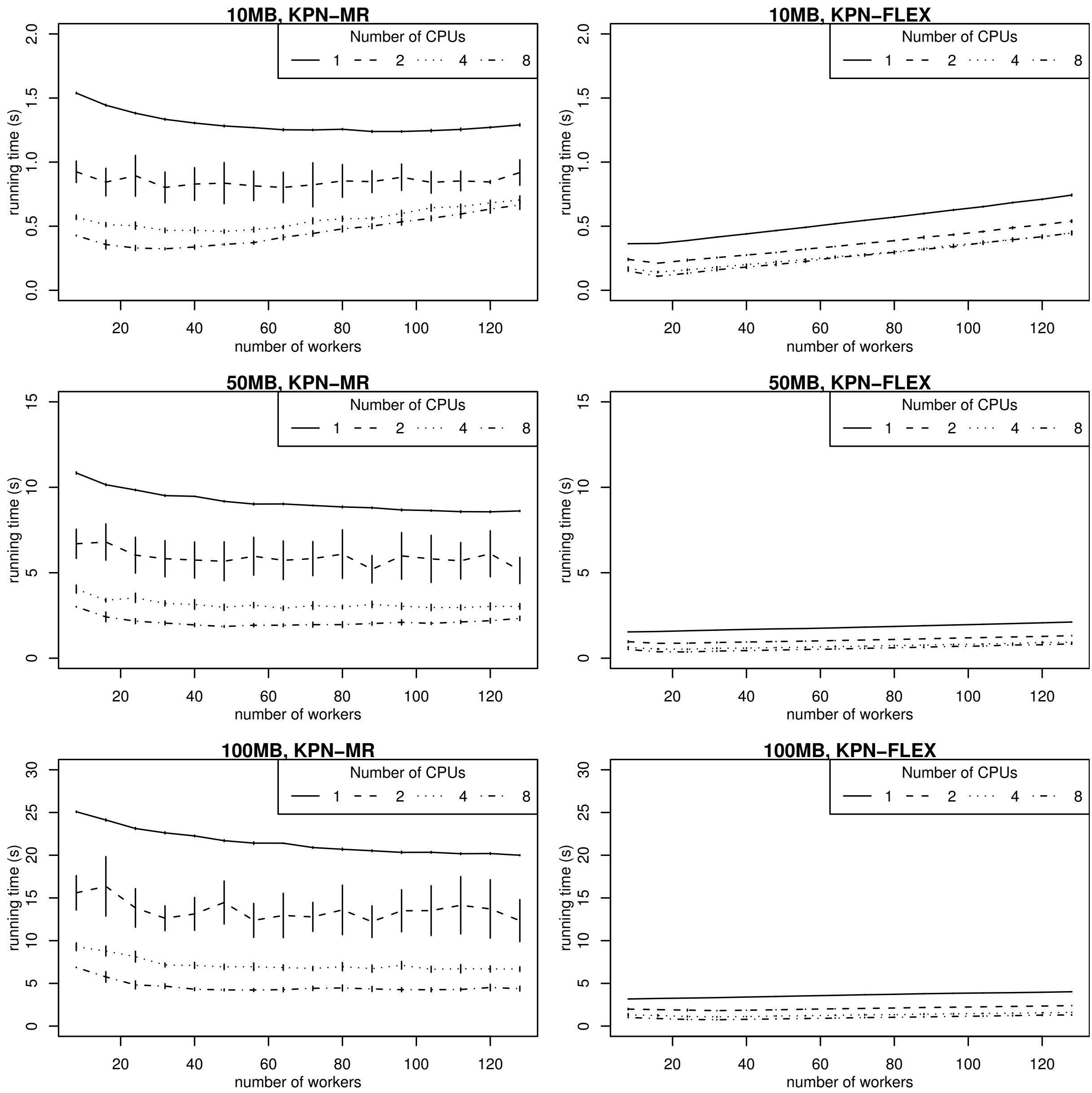


Figure 8. Running times for the word count program run on three different input files (10, 50 and 100 MB), implemented as KPN-FLEX and KPN-MR; vertical lines represent standard deviation. The number of workers for the KPN-MR version is *per stage*, so there are four times as many processes in total (two MR blocks, each consisting of two stages, each stage having the same number of workers).

ing the new means for the next iteration. Table 3 shows speedup of the KPN-FLEX solution over KPN-MR. We see that the speedup is ~ 1.2 (20%) for the benchmarks with 100 groups, and ~ 1.4 (40%) for the benchmark B with 50 groups. Since KPN-MR is recalculating means in parallel, this leads us to believe that the single I process which recalculates new means is the bottleneck in the KPN-FLEX solution. Nevertheless, unlike the word count example, the k-means program shows *almost perfect linear scalability* with the number of CPUs.

6.4 Comparison with Phoenix

We have also benchmarked our KPN-FLEX and KPN-MR against Phoenix [13], which is a MapReduce implementation designed specifically for multicore machines.

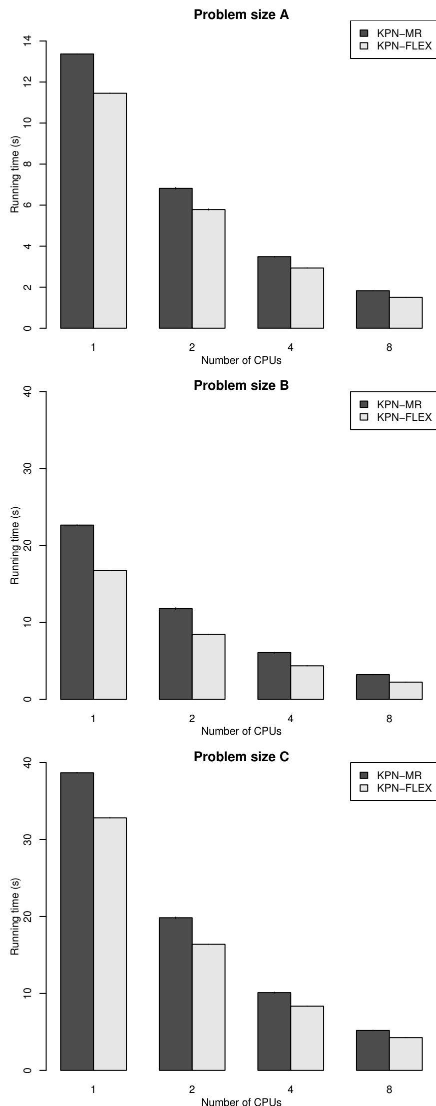


Figure 9. Benchmark results for the k-means program run on three different problem sizes, implemented as KPN-MR and KPN-FLEX. Vertical lines (barely visible) represent standard deviation. KPN-FLEX solution has as many worker process as the number of runners that were used in the benchmark. Similarly, the KPN-MR solution has the same number of workers as CPUs in each stage.

Phoenix is implemented with pthreads and uses by default as many threads in each of the Map and Reduce stages as there are CPUs on the machine. We had to compile Phoenix programs in 32-bit mode because their code uses non-portable casts between pointers and integers, which causes crashes in 64-bit mode. This is certainly to Phoenix's advantage because it allocates arrays of pointers, which would take twice as much space in the 64-bit mode, which causes fewer cache and TLB misses than in 64-bit mode.

Size	KPN-MR		KPN-FLEX		
	t	α	t	α	f
A/1	13.37	1.00	11.45	1.00	1.17
A/2	6.82	1.96	5.79	1.98	1.18
A/4	3.49	3.83	2.93	3.91	1.19
A/8	1.83	7.31	1.51	7.58	1.21
B/1	22.64	1.00	16.74	1.00	1.35
B/2	11.80	1.92	8.44	1.98	1.40
B/4	6.07	3.73	4.35	3.85	1.40
B/8	3.20	7.08	2.24	7.47	1.43
C/1	38.68	1.00	32.83	1.00	1.18
C/2	19.84	1.95	16.39	2.00	1.21
C/4	10.11	3.83	8.35	3.93	1.21
C/8	5.19	7.45	4.26	7.71	1.22

Table 3. k-means experiment summary: mean running time (t), relative speedup over one CPU (α) and and speedup factor of KPN-FLEX over KPN-MR (f).

Size	Phoenix	KPN-MR		KPN-FLEX	
	t (σ)	t	f	t	f
Word Count					
10	0.30 (0.02)	0.32	0.94	0.11	2.73
50	0.98 (0.02)	1.86	0.53	0.37	2.65
100	2.04 (0.05)	4.23	0.48	0.74	2.76
k-Means					
A	2.39 (0.05)	1.83	1.31	1.51	1.58
B	3.92 (0.21)	3.20	1.23	2.24	1.75
C	5.43 (0.22)	5.19	1.05	4.26	1.28

Table 4. Mean running times in seconds (t) of the word count and k-means programs for Phoenix, KPN-MR and KPN-FLEX; in case of KPNs, the smallest time on 8 CPUs is shown. σ is standard deviaton and f is speedup factor over Phoenix.

Table 4 shows Phoenix, KPN-MR, and KPN-FLEX running times for word count and k-means programs on all problem sizes. All Phoenix experiments have been run on the same machine 10 times, and the mean and standard deviation has been calculated.

In the word count benchmark, the KPN-MR program is consistently slower than Phoenix, by a factor of 1.06 on the 10MB file, and by a factor of ~ 2 on 50MB and 100MB files. This result is not very surprising since Phoenix is optimized for running MapReduce programs, while our KPN runtime assumes nothing about the network topology. However, the KPN-FLEX program, by having the ability to use data structures that are more suited to the given task (hash tables) and avoiding unnecessary work that MapReduce semantics requires (extra sort), achieves 2.6 times better performance than Phoenix.

Somewhat more surprisingly, both our KPN-MR and KPN-FLEX solutions outperform Phoenix on the k-means benchmark, by factors of 1.05 – 1.3 and 1.28 – 1.75, respectively. The KPN-FLEX solution performs better because it

avoids doing unnecessary work. However, we are as of yet unsure about the reasons for the unexpectedly good performance of the KPN-MR solution. We believe that the main reason for good performance of the KPN-MR solution is the elimination of framework (de)initialization that Phoenix has to perform on each iteration.

7 Conclusion and future work

In this paper we have reviewed in detail MapReduce and Dryad frameworks and KPNs. We have identified KPNs as the most flexible abstraction that presents a sequential programming model, while still ensuring deterministic execution of programs. We have also demonstrated that the MapReduce paradigm, widely recognized for its simplicity, is only a specially crafted KPN with fixed communication patterns.

To investigate advantages of KPN models, we have implemented the word count and k-means programs using two KPN topologies: one closely implementing the semantics of MapReduce (KPN-MR), and another which is specially crafted to the problem at hand (KPN-FLEX). In our benchmarks, on a 64-bit 8-core machine, KPN-FLEX implementation always outperforms the KPN-MR implementation (up to a factor of 1.3 for the k-means program and up to a factor of 6.7 for the word count program). Compared with Phoenix [13], KPN-FLEX implementations of word count and k-means programs are faster by a factor of up to 2.7 and 1.7. KPN-MR implementation of the word count program is about 2 times slower than Phoenix, but the KPN-MR implementation of the k-means program is, surprisingly, up to a factor of 1.3 faster than Phoenix.

Our future work includes performing more extensive tests to better understand performance characteristics of KPNs, especially in relation to the total number of processes and increasing scalability on machines with many CPUs. Work on scalability can be done by experimenting in three areas: using lock-free queues for channel communication (e.g., the one described in [8]), evaluating trade-offs between using lock-free queues or locks with exponential backoff in the work stealing scheduler, as well as implementing a distributed (as opposed to the current centralized) deadlock detection and resolution algorithm [1].

Acknowledgments

We thank to Håvard Espeland for constructive discussions during writing of this paper.

References

[1] G. Allen, P. Zucknick, and B. Evans. A distributed deadlock detection and resolution algorithm for process networks. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2:II–33–II–36, April 2007.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 119–129, New York, NY, USA, 1998. ACM.

[3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008.

[4] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of ACM international conference on Management of data (SIGMOD)*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[5] E. de Kock, G. Essink, W. J. M. Smits, R. van der Wolf, J.-Y. Brunei, W. Kruijtzer, P. Lieverse, and K. K.A. Vissers. Yapi: application modeling for signal processing systems. *Proceedings of Design Automation Conference*, pages 402–405, 2000.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *Programming Languages and Systems, European Symposium on Programming (ESOP)*, pages 319–334. Springer Berlin/Heidelberg, 2003.

[8] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP: Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52, New York, NY, USA, 2008. ACM.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, New York, NY, USA, 2007. ACM.

[11] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74, 1974.

[12] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[13] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

Paper VII

The Nornir run-time system for parallel programs using Kahn process networks

Zeljko Vrba, Pål Halvorsen, Carsten Griwodz, Paul B. Beskowand Dag Johansen

Published: Proceedings of 6th International Conference on Network and Parallel Computing (NPC), ed. by NA, pp. 1-8, IEEE Computer Society, 2009

Evaluation: 85 papers submitted, 35 accepted, with acceptance rate of 41.1%

Author contribution: Zvrba was the primary investigator and contributor to this paper. Beskow provided some auxiliary code, investigations into related work and to the discussion of ideas.

The *Nornir* run-time system for parallel programs using Kahn process networks

Željko Vrba, Pål Halvorsen, Carsten Griwodz, Paul Beskow
Simula Research Laboratory, Oslo, Norway
Department of Informatics, University of Oslo, Norway
{*zvrba,paalh,griff,paulbb*}@*ifi.uio.no*

Dag Johansen
Dept. of Computer Science, University of Tromsø, Norway
Fast Search and Transfer, Norway
{*dag@cs.uit.no*}

***Abstract*—Shared-memory concurrency is the prevalent paradigm used for developing parallel applications targeted towards small- and middle-sized machines, but experience has shown that it is hard to use. This is largely caused by synchronization primitives which are low-level, inherently nondeterministic, and, consequently, non-intuitive to use. In this paper, we present the *Nornir* run-time system. Nornir is comparable to well-known frameworks like MapReduce and Dryad, but has additional support for process structures containing cycles. It is based on the formalism of Kahn process networks, which we deem as a simple and deterministic alternative to shared-memory concurrency. Experiments with real and synthetic benchmarks on up to 8 CPUs show that performance in most cases improves almost linearly with the number of CPUs, when not limited by data dependencies.**

I. Introduction

It is widely recognized that developing parallel and distributed programs is inherently more difficult than developing sequential programs. As a consequence, several frameworks that aim to make such development easier have emerged, such as Google's MapReduce [1], Yahoo's PigLatin [2] which uses Hadoop[1] as the back-end, Phoenix [3], and Microsoft's Dryad [4]. All of these frameworks are gaining in popularity, but they lack a feature that is critical to our application domains: the ability to model iterative algorithms, i.e., algorithms containing feedback loops in their data-path.

Much of our research focuses on the execution of complex parallel programs, such as real-time encoding of 3-D video streams and data encryption, where cycles are more rule than the exception (see figure 2). Thus, we cannot use any of the existing frameworks, so we have turned towards the flexible formalism of Kahn process networks (KPN) [5]. KPNs retain the nice properties of MapReduce and Dryad, but in addition support cycles. Even though KPNs are an inherently distributed model of computation, their implementation for shared-memory machines and its performance is worth studying for many reasons (see section II), the main ones being determinism and, consequently, composability. Determinism guarantees that a program, given the same input, will behave identically on each run. This significantly eases debugging, which is an otherwise a notoriously hard problem with parallel and distributed computations. Composability guarantees that assembling together independently developed components will yield the expected result.

In our earlier paper [6], we have evaluated implementation options for KPNs on shared-memory architectures. In a follow-up paper [7], we have presented case studies of modeling with KPNs and their comparison with MapReduce. We showed that KPNs allow more natural problem modeling than MapReduce, and that implementations of real-world tasks on top of Nornir outperform the corresponding MapReduce implementations. In this paper, we give implementation details about the new version of Nornir, which is internally significantly different from the one described in [6]; this same implementation is also used for evaluation in [7]. We also investigate its performance and scalability characteristics of Nornir using a set of benchmarks on a workstation-class machine with 8 cores. Our performance experiments reveal some weaknesses in our current implementation, but nevertheless indicate that KPNs are a viable programming model for parallel applications on shared-memory architectures.

II. Kahn process networks

KPNs, MapReduce and Dryad have in common two important features, both of which significantly simplify development of parallel applications: 1) communication and parallelism are explicitly expressed in the application graph; 2) individual processes are written in the usual sequential manner, and do not have access to each other's state. In addition, KPNs have a unique combination of other desireable properties:

- *Determinism.* KPNs are deterministic, i.e., each execution of a network produces the same output given the same input,[2] regardless of scheduling strategy.
- *Reproducible faults.* One consequence of determinism is that faults are consistently reproducible, which is otherwise a notoriously difficult problem with parallel and distributed systems. Reproducibility of faults greatly eases debugging.

[1] An open-source MapReduce implementation in Java, available at http://hadoop.apache.org/

[2] Provided that processes themselves are deterministic.

- *Composability.* Another consequence of determinism is that processes can be composed: connecting the output of a process computing $f(x)$ to the input of a process computing $g(x)$ is guaranteed to compute $g(f(x))$. Therefore, processes can be developed and tested individually and later put together to perform more complex tasks.
- *Deterministic synchronization.* Synchronization is embodied in the blocking receive operation. Thus, developers need not use other, low-level and non-deterministic synchronization mechanisms such as mutexes and condition variables.[3]
- *Arbitrary communication graphs.* Whereas MapReduce and Dryad restrict developers to a parallel pipeline structure and directed acyclic graphs (DAGs), KPNs allow *cycles* in the graphs. Because of this, they can directly model iterative algorithms. With MapReduce and Dryad this is only possible by manual iteration, which incurs high setup costs before each iteration [3].
- *No prescribed programming model.* Unlike MapReduce, KPNs do not require that the problem be modeled in terms of processing over key-value pairs. Consequently, transforming a sequential algorithm into a Kahn process often requires minimal modifications to the code, consisting mostly of inserting communication statements at appropriate places.

A KPN [5] has a simple representation in the form of a *directed graph* with *processes* as nodes and *channels* as edges, as exemplified by figure 1. A process encapsulates data and a single, sequential control flow, independent of any other process. Processes are not allowed to share data and may communicate only by sending messages over channels. Channels are *infinite* FIFO queues that store discrete *messages*. Channels have *exactly one* sender and *one* receiver process on each end (1:1 communication), and every process can have multiple *input* and *output* channels. Sending a message to the channel always succeeds, but trying to receive a message from an empty channel *blocks* the process until a message becomes available. These properties define the *operational semantics* of KPNs and make the Kahn model *deterministic*, i.e., the history of messages produced on the channels does not depend on how the process execution order. Less restrictive models, e.g., those that allow non-blocking reads or polling, would result in non-deterministic behavior.

The theoretical model of KPNs described so far is idealized in two ways: 1) it places few constraints on process behavior, and 2) it assumes that channels have infinite capacities. These assumptions are somewhat problematic because they allow for the construction of KPNs that need

[3]Many inexperienced developers expect that mutexes and CVs wake up waiting threads in FIFO order, whereas the wake-up order is in reality non-deterministic.

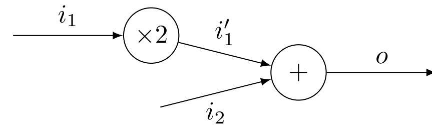


Figure 1. An example KPN. i_1 and i_2 are external input channels to the network (assumed to be numbers), o is the external output channel, and i'_1 is an internal channel. The inputs and the output are related by the formula $o = 2i_1 + i_2$

unbounded space for their execution. However, any real implementation is constrained to run in finite memory. A common (partial) solution to this is to assign *capacities* to channels and redefine the semantics of send to *block* the sending process if the delivery would cause the channel to exceed its capacity. Under such send semantics, an *artificial deadlock* may occur, i.e., a situation where a cyclically dependent subset of processes blocks on send, which would continue running in the theoretical model. The algorithm of Geilen and Basten [8] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarging it by one message, thus resolving the deadlock.

It also is worth noting that KPNs are not a universal solution for what is an inherently difficult problem of developing parallel and distributed applications. Even though determinism is a desirable property from the standpoint of program design and debugging, it limits the applications for KPNs. A disk scheduler serving many clients is a very simple example of a use-case that is inappropriate for KPNs. It must periodically serve all clients in some order to preserve fairness, say round-robin, but since read is blocking, absence of requests from one client can indefinitely postpone serving of requests from other clients. Such use-cases mandate use of other frameworks, or extending the KPN formalism by non-deterministic construct(s) such as $m : n$ channels and/or polling.

III. Nornir

The Nornir run-time system is implemented in C++, and it runs on Windows and POSIX operating systems (Solaris, Linux, etc.). The implementation[4] consists of a Kahn process (KP) scheduler, message transport and deadlock detection and resolution algorithms.

A. Process scheduler

Since KPNs are deterministic, they give a great freedom in implementing the process scheduler: any *fair* scheduler will result in a KPN execution that generates the full output.[5] In this context, fairness means that the execution of a ready process will not be indefinitely postponed.

[4]Code is available at: http://simula.no/research/networks/software

[5]If the scheduler is not fair, the output will be correct, but possibly shorter than it would be under a fair scheduler.

KPN networks could be built on top of native OS mechanisms, with each KP being an OS-thread. Channels would be protected by blocking mutexes, and condition variables would be used as the sleep / wakeup mechanism. However, we have investigated this approach in an earlier paper [6] and found that user-mode scheduling of many KPs over few kernel threads is considerably more efficient.

Nornir can be configured to use different scheduling policies. In addition to classical work-stealing [9], we have also implemented a policy based on graph-partitioning [10] which tries to reduce the amount of inter-CPU synchronization and communication. We describe here only the work-stealing policy due to its superiority and space restrictions.

When the KPN is started, m runner threads ("runners") are created and scheduled by the OS onto the available CPUs. Each runner implements a work-stealing policy, which our experiments have shown to have best performance for computationally intensive tasks on few CPUs. With work stealing, each runner has a private run queue of ready KPs. If this queue is empty, it tries to steal a KP from a randomly chosen runner. For simplicity, we do not use the non-blocking queue described in [9]; instead we use an ordinary mutex per run queue. This might become problematic on machines with many cores, but we deemed that introducing the additional complexity of a non-blocking queue was unnecessary at this stage of our research.

The work-stealing scheduler uses a user-mode context-switch. On Solaris and Linux running on AMD64 architecture, we employ optimized, hand-crafted assembly code for context-switch; on other platforms we use OS-provided facilities: fibers on windows, and `swapcontext()` on POSIX. The latter is inefficient because each context switch requires a system call to save and restore the signal mask.

B. Message transport

In KPNs, channels have a two-fold role: 1) to interact with the scheduler, i.e., block and unblock processes on either side of the channel, and 2) to transport messages. The initial capacity of the channel may be specified when the channel is first created; if omitted, the default capacity taken from an environment variable is used.

Interaction with the scheduler is needed for the cases of a full or empty channel. Receiving from an empty channel or sending to a full channel must block the acting process. Similarly, receiving from a full channel or sending to an empty channel must unblock the process on the other side of the channel.

Message transport over channels is protected by *busy-wait* mutexes: if a KP cannot obtain the channel's lock, it will explicitly yield to the scheduler between iterations, until it has finally obtained the lock. Busy-waiting allows other processes to proceed with their computations, while avoiding the complexities of a full-fledged sleep/wakeup mechanism. Furthermore, since channels are 1:1, at most two processes will compete for access to any given channel, so the expected number of spins in the case of contention on a channel is very small.

Since KPs are executing in a shared address space in our implementation, it is still possible that they modify each others state[6] and thus ruin the KPN semantics. There are at least two ways of implementing a channel transport mechanism that lessens the possibility of such occurrence:

- A message can be dynamically allocated and a pointer to it sent in a class that implements *move semantics* (e.g., `auto_ptr` from the C++ standard library).
- A message can be physically copied to/from channel buffers which is, in our case, done by invoking the copy-constructor.

We have initially implemented the first approach, which requires dynamic memory (de-)allocation for every message creation and destruction, but is essentially zero-copy. Our current implementation uses the second approach because measurements on Solaris have shown that memory (de)allocation, despite having been optimized by using Solaris's *umem* allocator, has larger overhead than copying as long as the message size is less than ~ 256 bytes. As of now, our implementation cannot choose between different mechanisms depending on the message size, so we recommend that large blocks be transfered as pointers.

Since C++ is a statically-typed language, our channels are also *strongly-typed*, i.e., they carry messages of only a single type. Since *communication ports* (endpoints of a channel; used by processes to send and receive messages) and channels are parametrized with the type of message that is being transmitted, compile-time mechanisms prevent sending messages of wrong types. Furthermore, the run-time overhead of dynamic dispatch based on message type are eliminated. Nevertheless, if dynamic typing is desired, it can be implemented by sending byte arrays over channels, or in a more structured and safe manner by using a standard solution such as *Boost.Variant* (see http://www.boost.org).

As KPs have only blocking read at their disposal, it is useful to provide an indication of no more messages arriving on the channel (EOF). One way of doing this is to send a message with specific value that will indicate EOF. However, all values of a type (e.g., `int`) might be meaningful in a certain context, so no value is available to encode the EOF indication. In such cases, one would be forced to use solutions that are more cumbersome to use and impose additional overhead (for example, dynamic memory allocation with `NULL` pointer value representing EOF). We have therefore extended channels by introducing support for *EOF indication*: the sender can set the EOF status on the channel when it has no more messages to send. After EOF on the channel has been set, the receiver will be able to

[6] C++ is an inherently unsafe language, so there is no way of *preventing* this.

read the remaining buffered messages. After all messages have been read, the next call to the port's `recv` method will immediately return false (without changing the target message buffer), and the next `recv` call will permanently block the process.

C. Deadlock detection and resolution

Deadlock detection and resolution makes it possible to execute KPNs in a finite space. Each time a process would block, either on read or on write, a deadlock detection routine is invoked. Since communication is 1:1, every cycle of blocked KPs is a ring; a property which greatly simplifies detection. The deadlock detection and resolution algorithm in our current implementation uses a centralized data-structure (the blocking graph) and thus must run while holding a single global mutex. If no cycle is found, the KP is blocked and this fact is recorded in the blocking graph. Otherwise, the capacity of the smallest channel in the cycle is increased by one, as suggested by [8]. Similarly, receiving from a full channel unblocks the sending side and removes the corresponding edge from the blocking graph.

D. Accounting

We have implemented a detailed accounting system that enables us to monitor many different aspects of Nornir's run-time performance, such as cpu time used by each process, number of context-switches, number of loop iterations in waiting on spinlocks, number of process thefts, number of messages sent to processes on the same or different CPU. We have measured (see [6] for methodology) that a single transaction consisting of [send $\rightarrow$ context switch $\rightarrow$ receive] takes $1.4\mu s$ with accounting enabled. When accounting is disabled, this time drops to $\sim 0.68\mu s$. The largest overhead in our accounting mechanism stems from the measurement of per-process CPU time, which requires a system call immediately before a process is run and immediately after a process returns to scheduler.

IV. Case study: H.264 encoding

KPNs are especially well suited for stream-processing applications. Indeed, each process is a function that takes as input one or more input data streams and produces one or more streams as output. H.264 is a modern, lossy video-compression format that offers high compression rates with good quality. Our KPN representation of an H.264 video encoder (see figure 2) is a slight adaptation of the encoder block diagram found in [11], with functional blocks implemented as KPs.

The input video consists of a series of discrete frames, and the encoder operates on small parts of the frame, called macroblocks, typically 16×16 pixels in size. The encoder consists of "forward" and "reconstruction" datapaths which meet at the prediction block (P). The role of the prediction block is to decide whether the macroblock will be encoded by using intra-prediction (relative to macroblocks in the *current* frame F_n) or inter-prediction (relative to macroblocks in the *reference* frame(s) F_{n-1}). The encoded macroblock goes through the forward path and ends at the entropy coder (EC), which is the final output of the encoder. The decision on whether to apply intra- or inter-prediction is based on factors such as the desired bandwidth and quality. To be able to estimate quality, the codec needs to apply transformations inverse to those of the forward path, and determine whether the *decoded* frame satisfies the quality constraints.

This example demonstrates that feedback loops are not only a matter of *convenience*, but actually *essential* for the expressive power of a programming framework. Thus, as already argued in the introduction, neither MapReduce nor Dryad can be used to implement the H.264 encoder.

V. Performance evaluation

To evaluate the performance and scalability of applications modeled as KPNs, we have used the H.264 KPN network (see section IV and Figure 2), a random network and a pipeline with artificial workloads, as well as AES encryption with a real workload. The test programs have been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). All benchmarks have been configured to use the work-stealing scheduling policy, initial channel capacity of 64 messages. Nornir has been compiled with accounting turned on, since this is necessary to study performance effects of deadlock detection. We have run them on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM running Linux kernel 2.6.27.3. Each data point is an average of 10 consecutive measurements of the total real (wall-clock) running time. This metric is most representative because it accurately reflects the real time needed for task completion, which is what the end-users are most interested in.

A. Results

H.264: For the purpose of evaluation of Nornir, we have used an artificial workload consisting of loops that consume the amount of CPU time which are on average used by a real codec in the different blocks. To gather this data, we have profiled x264, an open-source H.264 encoder, with the cachegrind tool and mapped the results to the H.264 block-diagram (see figure 2). The benchmark "encoded" 30 frames at rate of 1 frame per second (fps), with the number of workers varying from 1...512 in successive powers of two. From the results in figure 3 we can see that the performance gets slightly better as the number of workers per stage increases up to the number of CPUs, and remains constant afterwards. The best achieved speedup on 8 CPUs is only ~ 2.8; this limitation is caused by data-dependencies in the algorithm.

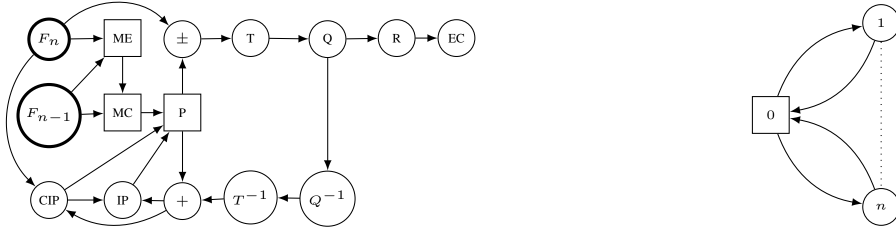


Figure 2. H.264 block-diagram, adapted from the H.264 whitepaper [11]. Inputs to the codec are current and reference frames (F_n and F_{n-1}). Since P, MC and ME blocks are together using over 50% of the total processing time and are perfectly parallelizable, we have parallelized each block by dividing the work over n workers. The figure on the right exemplifies parallelization of a single block.

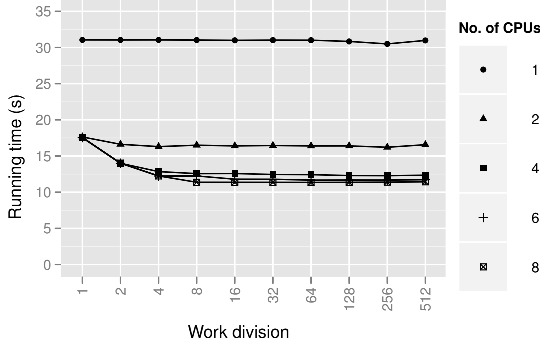


Figure 3. H.264: performance of "encoding" 30 frames at ~ 1 fps.

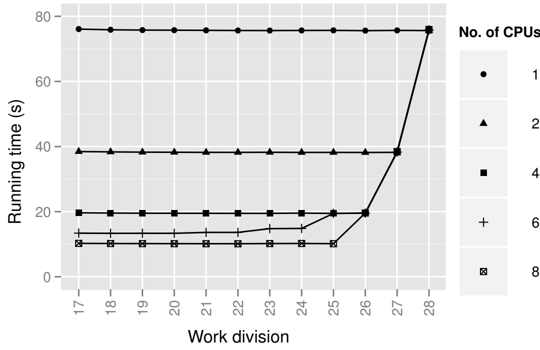


Figure 4. AES encryption benchmark.

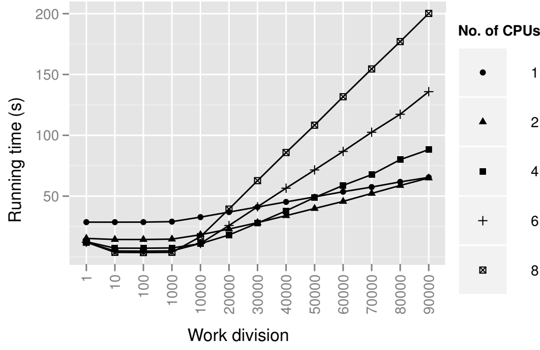
AES: The AES KPN has the same topology as the network on the right in figure 2; this topology is in fact common for algorithms that can process different chunks of input independently. The source KP (denoted "0") hands out equally-sized memory chunks to n worker KPs (denoted "1" …"n") which reply back when they have encrypted the given chunk. The exchanged messages are carrying only pointer-length pairs (16 bytes in total).

In this benchmark, we have set the total block size to 2^{28} bytes (256 MB), and the chunk given to each individual worker has been varied from to 2^{17} to 2^{28} and the total number of workers has been varied from 2048 to 1. The number of encryption passes that each worker will perform over its assigned chunk has been set to 40. The results, shown in figure 4, show perfect linear speedup with the number of CPUs, as soon as the number of workers becomes greater or equal to the number of CPUs. Note that the number of workers increases to the *left* in the figure, when given work division w (x-axis), the number of workers is 2^{28-w}. For $w = 28$ there cannot be any speedup on multiple CPUs because there is only a single worker process encrypting the whole chunk.

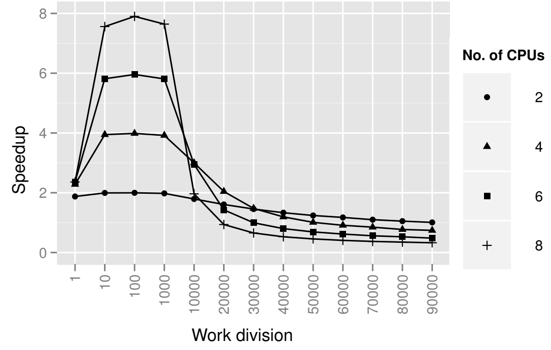
Random network: A random network is a directed graph consisting of a source KP, a number of intermediate KPs arranged in n_l layers (user specified) and a sink KP. The number of KPs in each layer is randomly selected between 1 and the user-specified maximum number m. The intention of this construction is to mimic, with fine granularity, network protocol graphs or parallelism in algorithms. The network may have additional b back-edges which create cycles. Each node is constrained to have at most one back-edge, be it outgoing or incoming.

The workload of the random network is determined by the formula nT/d, where n is the number of messages sent by the source, T is a constant that equals ~ 1 second of CPU-time, and d is the work division factor. In effect, each single message sent by the source (a single integer) carries a work amount equalling approximately $w = T/d$ seconds of CPU time. The workload w is distributed in the network (starting from the source KP) with each KP reading n_i messages from all of its in-edges. Once all messages are read, they are added together to become the t units of CPU-time which the KP is to consume before distributing t to its n_o forward out-edges. Then, if a process has a back-edge, a message is sent/received (depending on the edge direction) along that channel. As such, the workload w distributed from the source KP will equal the workload w collected by the sink KP. Messages sent along back-edges do not contribute to the network's workload; their purpose is solely to generate more complex synchronization patterns.

Figure 5(a) shows the absolute running times of a random



(a) Running time.



(b) Speedup over 1 CPU.

Figure 5. Benchmark of a random directed graph with 212 nodes, 333 edges and 13 cycles in 50 layers. The x-axis is not uniform.

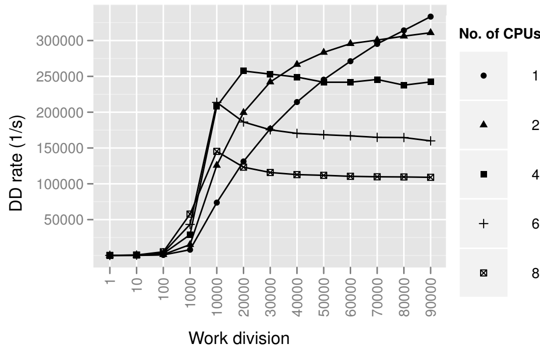


Figure 6. Deadlock detection rate on the random graph benchmark.

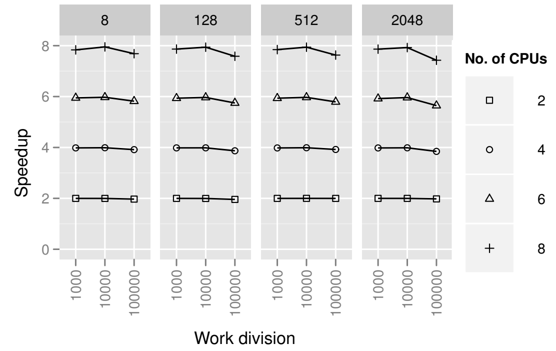


Figure 7. Pipeline speedup for message sizes of 8,...,2048 bytes and three different work divisions (d).

graph with cycles at different values of d, whereas figure 5(b) shows speedup over running time on 1 CPU. In our experiments, we have set $n = d$. As d increases to 100, the available parallelism in the network increases, and the running time decreases; in figure 5(b) we see that $d = 100$ achieves perfect speedup on multiple CPUs. At $d = 1000$, running time starts increasing linearly with d, and grows faster on more CPUs. This is caused by frequent deadlock detections, as witnessed by figure 6 which shows the number of started deadlock detections per second of running time. Since deadlock detection uses a global lock to protect the blocking graph, this limits Nornir's scalability on this benchmark. A possible way of avoiding this problem, in our current implementation, is to increase default channel capacity to a larger value. A long-term solution is implementing a distributed deadlock detection and resolution algorithm [12].

Pipeline: A pipeline does the same kind of processing as the random network, except that each layer has exactly one process and each process takes its only input from the preceding process in the pipeline. In all previous benchmarks, the communicated messages have been rather small (less than 32 bytes). We have used a pipeline consisting of 50 stages to study the impact of message size on performance. As previously, d messages have been generated by the source process, each containing $1/d$ seconds of CPU time. A noticable slow-down (see figure 7) happens regardless of message size and only at $d = 10^5$, which is equivalent to 10 μs of work per message, which is only 7 times greater than the time needed for a single transaction (see section III-D). As expected, the drop in performance at $d = 10^5$ is proportional to message size, but also to the number of CPUs. Larger message sizes take more time to copy, which causes greater contention over channel locks with increasing number of CPUs.

B. Summary

We have evaluated several aspects of Nornir: scalability of the scheduler with number of processes and CPUs,

overheads of copying message-passing and overheads of centralized deadlock-detection and resolution. Our findings can be summarized as follows:

- Nornir can efficiently handle a large number of processes. Indeed, in the AES benchmark, it achieved an almost perfect linear speedup of 7.5 on 8 CPUs with 2048 processes.
- Message sizes up to 512 bytes have negligible impact on performance. The cost of message copying starts to be noticeable at message size of 2048 bytes. Protecting channels with mutexes has negligible performance impact on 8 CPUs.
- As shown by the pipeline benchmark, context-switch and message-passing overheads start to have a noticeable impact on the overall performance when the amount of work per message is less than ~ 7 times the transaction time (see section III-D).
- The centralized deadlock detection and resolution algorithm can cause serious scalability and performance problems on certain classes of applications. In our evaluation, this was the case *only* for the random graph benchmark.
- Again, as shown by the random graph benchmark, the default channel capacity of 64 bytes, which we have used in our benchmark, can be too small in certain casses. Increasing it would mitigate overheads of deadlock detection, but it would also increase memory consumption.
- Performance can be further increased by turning off detailed accounting in cases where it is not needed.
- We have not noticed any scalability problems with using mutexes to protecting the scheduler's queues instead of using the non-blocking queue of [9].

Although there is room for improvement in Nornir (especially in deadlock detection), our results indicate that message-passing and KPNs in particular are a viable programming model for high-performance parallel applications on shared-memory architectures.

VI. Related work

Very few general-purpose KPN runtime implementations exist, among them YAPI [13] and Ptolemy II [14]. YAPI is not a pure KPN implementation, as it extends the semantics and thus introduces the possibility of non-determinism, its code-base is too large for easy experimentation (120 kB vs. 50 kB in our implementation), and the implementation did not have inherent support for multiple-CPUs. Ptolemy II is a Java-based prototyping platform for experimenting with various models of computation, and it spawns one thread for each Kahn process, which is rather inefficient for large networks. The amount of code that the JVM consists of would make it prohibitively difficult to experiment with low-level mechanisms, such as context-switches. PNRunner, a part of the Sesame project [15], is an event-driven *simulator* of embedded systems, which employs KPNs for application modeling and simulation. As such, it is not suitable for executing KPNs where performance is important.

Phoenix [3] is a MapReduce implementation optimized for multi-core architectures. We have reimplemented word count and k-means examples using Nornir, and found that our implementation outperforms that of Phoenix by factors of up to 2.7 and 1.7, respectively [7]. The main reason for this is that, unlike MapReduce, KPNs allow us to use algorithms and build a processing graph that are well matched to the structure of the underlying problem.

StreamIt [16] is a language for simplifying implementation of stream programs described by a graph consisting of computational blocks (filters) having a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous dataflow process network [17]. The compiler produces code which can exploit multiple machines or CPUs, but their number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

PigLatin [2] is a language for performing ad-hoc queries over large data sets. Users specify their queries in a high-level language which provides many features of operators found in SQL. Unlike SQL, which is declarative and heavily relies on query optimizer for efficient query execution, Pig latin allows users to precisely specify *how* the query will be executed. In effect, users are constructing a dataflow graph which is then compiled into a pipeline of MapReduce programs and executed on a Hadoop cluster, which is an open-source, scalable implementation of MapReduce. All of the Pig latin operators, such as `FILTER` and `JOIN`, are directly implementable as Kahn processes. Taking our experimental results [7] into consideration, we believe that compiling Pig latin programs into Nornir graphs would be advantageous for their performance on multi-core machines.

VII. Conclusion and future work

In this paper we have described implementation details of Nornir, our run-time environment for executing parallel applications specified in the high-level framework of Kahn process networks, which allow cycles in the communication graph of the program. Since this feature is crucial for implementing iterative algorithms such as H.264 encoding, Nornir complements existing frameworks such as MapReduce and Dryad.

We have evaluated Nornir's efficiency with several synthetic (H.264 encoding, random KPN, pipeline) and one real (AES) application on an 8-core machine. Our results indicate that Nornir can scale well, but that in certain cases (random KPN) the centralized deadlock detection is detrimental for performance, and that default channel capacity of 64 bytes is too small for some applications. We have also found that copying semantics of message-passing starts

having a slight, but noticeable impact on performance at message sizes of ~ 2048 bytes. Furthermore, Nornir can support parallelism at fine granularity: its overheads become noticeable at processing time of $10\mu s$ per message, which is ~ 7 times greater than the combined overhead of scheduling and message-passing.

The first, and most important, step in our future work is increasing Nornir's scalability by replacing a centralized deadlock detection algorithm with a a distributed one [12]. This will also be the first step towards a distributed version of Nornir, executing on a cluster of machines. Further performance increases can be gained by using non-blocking data structures. In the scheduler, we might need to use the non-blocking queue of [9] instead of mutexes in order to support scalability beyond 8 CPUs. We might also use a single-producer, single-consumer FIFO queue [18] to avoid yielding between en-/dequeue attempts. Since yielding incurs switching to new control flow and new stack, we expect that this improvement will further increase performance by reducing pressure on data and instruction caches.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of Symposium on Opearting Systems Design & Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*. New York, NY, USA: ACM, 2007, pp. 59–72.

[5] G. Kahn, "The semantics of a simple language for parallel programming." *Information Processing*, vol. 74, 1974.

[6] Željko Vrba, P. Halvorsen, and C. Griwodz, "Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors," in *Proceedings of the International Workshop on Multi-Core Computing Systems (MuCoCoS)*, 2009.

[7] Željko Vrba, P. Halvorsen, C. Griwodz, and P. Beskow, "Kahn process networks are a flexible alternative to mapreduce," in *To appear in: Proceedings of the International Conference on High Performance Computing and Communications*, 2009.

[8] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," in *Programming Languages and Systems, European Symposium on Programming (ESOP)*. Springer Berlin/Heidelberg, 2003, pp. 319–334.

[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA)*. New York, NY, USA: ACM, 1998, pp. 119–129.

[10] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, also available as Sandia National Labs Tech Report SAND2006-6450C.

[11] I. E. G. Richardson, "H.264/mpeg-4 part 10 white paper," Available online., http://www.vcodex.com/files/h264_overview_orig.pdf.

[12] G. Allen, P. Zucknick, and B. Evans, "A distributed deadlock detection and resolution algorithm for process networks," *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 2, pp. II–33–II–36, April 2007.

[13] E. de Kock, G. Essink, W. J. M. Smits, R. van der Wolf, J.-Y. Brunei, W. Kruijtzer, P. Lieverse, and K. K.A. Vissers, "Yapi: application modeling for signal processing systems," *Proceedings of Design Automation Conference*, pp. 402–405, 2000.

[14] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to Ptolemy II)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-28, Apr 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html

[15] M. Thompson and A. Pimentel, "Towards multi-application workload modeling in sesame for system-level design space exploration," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 4599/2007, 2007, pp. 222–232. [Online]. Available: http://www.springerlink.com/content/u4265u6r0u324215/

[16] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 151–162.

[17] T. Lee, E.A.; Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[18] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *PPoPP: Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 43–52.

Paper VIII

The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce

Zeljko Vrba, Pål Halvorsen, Carsten Griwodz, Paul B. Beskow, Håvard Espelandand Dag Johansen

Published: The Journal of Supercomputing, Springer(Online first):1-27, 2010

Evaluation: This journal paper was submitted by invitation to a Special Issue of Springer's The Journal of Supercomputing

Author contribution: Vrba was the primary investigator and contributor to this publication. Beskow provided investigations into related work and knowledge of distributed processing systems and discussion on ideas.

J Supercomput
DOI 10.1007/s11227-010-0503-2

The *Nornir* run-time system for parallel programs using Kahn process networks on multi-core machines—a flexible alternative to MapReduce

Željko Vrba · Pål Halvorsen · Carsten Griwodz · Paul Beskow · Håvard Espeland · Dag Johansen

© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract Even though shared-memory concurrency is a paradigm frequently used for developing parallel applications on small- and middle-sized machines, experience has shown that it is hard to use. This is largely caused by synchronization primitives which are low-level, inherently non-deterministic, and, consequently, non-intuitive to use. In this paper, we present the *Nornir* run-time system. Nornir is comparable to well-known frameworks such as MapReduce and Dryad that are recognized for their efficiency and simplicity. Unlike these frameworks, Nornir also supports process structures containing branches and cycles. Nornir is based on the formalism of Kahn process networks, which is a shared-nothing, message-passing model of concurrency. We deem this model a simple and deterministic alternative to shared-memory concurrency. Experiments with real and synthetic benchmarks on up to 8 CPUs show that performance in most cases scales almost linearly with the number of CPUs, when

Ž. Vrba (✉) · P. Halvorsen · C. Griwodz · P. Beskow · H. Espeland
Department of Informatics, University of Oslo, Oslo, Norway
e-mail: zvrba@ifi.uio.no

P. Halvorsen
e-mail: paalh@ifi.uio.no

C. Griwodz
e-mail: griff@ifi.uio.no

P. Beskow
e-mail: paulbb@ifi.uio.no

H. Espeland
e-mail: haavares@ifi.uio.no

Ž. Vrba · P. Halvorsen · C. Griwodz · P. Beskow · H. Espeland
Simula Research Laboratory, Oslo, Norway

D. Johansen
Department of Computer Science, University of Tromsø, Tromsø, Norway
e-mail: dag@cs.uit.no

Published online: 13 November 2010

not limited by data dependencies. We also show that the modeling flexibility allows Nornir to outperform its MapReduce counterparts using well-known benchmarks.

Keywords Parallel processing · Kahn process networks

1 Introduction

The introduction of commodity multi-core processors has spawned a wide interest in parallel programming. It is, however, widely recognized that developing parallel and distributed programs is inherently more challenging than developing sequential programs. As a consequence, several frameworks that aim to make such development easier have emerged, such as Google's MapReduce [12], Yahoo's PigLatin [28] which uses Hadoop [2] as the back-end, Phoenix [31] and Microsoft's Dryad [21]. All of these frameworks are gaining popularity, but they lack a feature that is critical to our application domains: the ability to model branching and iterative algorithms, i.e., algorithms containing feedback loops in their data-path.

Much of our research focuses on the execution of complex parallel programs, such as real-time encoding of 3-D video streams, where cycles are more the rule than the exception (see Fig. 6 for an example). Thus, we cannot use any of the existing frameworks, so we have turned towards the flexible formalism of Kahn process networks (KPN) [22]. KPNs retain the nice properties of MapReduce and Dryad, but in addition support cycles. Even though KPNs are an inherently distributed model of computation, their implementation for shared-memory machines and its performance is worth studying for many reasons, the main ones being determinism and composability. Determinism guarantees that a program, given the same input, will behave identically on each run. This significantly eases debugging, which is otherwise a notoriously hard problem with parallel and distributed computations. Composability guarantees that assembling independently developed components yields the expected results.

In an earlier paper [38], we evaluated implementation options for KPNs on shared-memory architectures. In a follow-up paper [39], we presented case studies of problem modeling with KPNs and compared the resulting KPN models with MapReduce models. We showed that KPNs allow more natural problem modeling than MapReduce, and that implementations of real-world tasks on top of Nornir outperform the corresponding MapReduce implementations optimized for multi-core machines. This paper expands our previous paper [40] with more detailed descriptions and further performance experiments. The implementation details of Nornir, which we describe in this paper, are somewhat different than the implementations of our previous KPN run-times described in [38, 39]. Most notably, we have added support for pluggable scheduling policies to allow easy experimentation. We also investigate the performance and scalability characteristics of Nornir using a set of benchmarks on a workstation-class machine with 8 cores. The experiments reveal some weaknesses of our current implementation, but indicate nevertheless that KPNs are a viable programming model for parallel applications on shared-memory architectures.

The rest of this paper is organized as follows. In the next section, we present a brief overview of related work. In Sect. 3, we present the properties and basic ideas behind

KPNs, and follow up with the description of Nornir implementation details in Sect. 4. In Sect. 5, we describe the benchmarks we designed and report our performance results. We summarize the paper and present concluding remarks in Sect. 6.

2 Background and related work

A lot of research has been dedicated to addressing the challenge of parallel and distributed programming, which has led to the development of many tools, programming languages and frameworks. Here, we give a short overview of this work—for a more detailed summary, please refer to [37].

2.1 Low-level tool support

Low-level tools can be roughly categorized into those enabling shared-state concurrency and those enabling message-passing concurrency. Shared-state concurrency has traditionally been most easily accessible from imperative programming languages such as C, C++ and FORTRAN. These languages have little built-in support for concurrency, so the developers must use the operating system's threading facilities or 3rd-party libraries, such as threading building blocks [20], which encapsulate common concurrency patterns. Another choice is OpenMP [33], which is an extension of C, C++ and FORTRAN oriented mostly towards loop-level parallelism. μC++ [6] is another extension of the C++ programming language, which introduces a number of concurrent programming concepts, such as cooperatively-scheduled co-routines [23] and tasks that run in parallel and are scheduled preemptively.

Message-passing concurrency has traditionally been used for implementing programs running in different address spaces or on different machines. Erlang [3] is an example of a pure functional programming language with built-in support for message-passing concurrency. In Erlang, processes (actors) communicate by sending messages via mailboxes, and the run-time provides extensive support for failure detection and recovery.

In most other languages, message-passing concurrency is delegated to library support, the most prominent examples of which are parallel virtual machine (PVM) and message passing interface (MPI). PVM [30] aims at support for heterogeneous concurrent computing, creating an illusion of a distributed virtual machine. It thus provides means for process and resource control and fault-tolerance along with communication primitives. MPI [26], on the other hand, is somewhat narrower in scope than PVM. Its main task is to support efficient point-to-point communication and some complex collective operations. Data types may be automatically serialized for network transport, and newer MPI versions have added many new features like dynamic process management and parallel I/O.

The tools described above provide sufficient *mechanisms* for building distributed and parallel applications, but they leave the definition of higher-level abstractions and policies to developers. For example, PVM and MPI enable inter-task communication, but do not endorse any particular application structure or concurrency model. In this respect, large players (Google and Microsoft in particular) have identified common patterns of distributed computations and packaged them in easier-to-use frameworks, which we describe in the next section.

2.2 High-level frameworks

Industrial actors have developed solutions for simplified distributed processing of large data quantities. Examples are Google's MapReduce [12], Yahoo's PigLatin programming language [28], Microsoft's Dryad [21] and Cosmos systems as well as the programming language Scope [8], and IBM's System S and programming language SPADE [14]. Dryad, Cosmos and System S have many properties in common: all use directed graphs to model computations and execute them on a cluster of machines. In addition, System S supports cycles in graphs, while Dryad supports non-deterministic constructs. Thus, the deterministic subset of the Dryad system is also a subset of the KPN framework, while the expressiveness of System S is equivalent to that of KPNs. However, not much is known about these systems and their availability is limited.

MapReduce [12] has become one of the most cited paradigms for expressing parallel computations. Unlike the above systems, which define task-parallel models, MapReduce defines a data-parallel model based on keys and values. While the original MapReduce paper specifies the programming model rather informally, Lämmel [24] has used the Haskell [19] programming language to rigorously define the semantics of the MapReduce model and Sawzall programming language [29]; there he also discusses parallelization issues. Google's MapReduce implementation supports fault-tolerant distributed execution in clusters. Others have re-implemented MapReduce for clusters (Hadoop [2], an open-source implementation in Java), multi-core machines [31], the Cell BE architecture [11] and even for GPUs [18], all of which bear witness of its popularity. Map-Reduce-Merge [9], adds a merge step to efficiently process data relationships among heterogeneous data sets, and to be able to execute in parallel join algorithms of relational algebra, operations not directly supported by the plain MapReduce model. The same issues are also addressed by Oivos [35], which in addition provides a more expressive, declarative programming model. Finally, reducing the added overhead of layering software on top of MapReduce is the goal of Cogset [36] where the processing architecture is changed to increase performance.

In our work, however, we are generally interested in real-time processing of multimedia content. This requires high performance and repeated non-trivial sequences of processing steps, but the above frameworks are mostly targeted towards off-line batch processing of data. MapReduce, Dryad and Cosmos are not able to model iterative algorithms; in addition the rigid MapReduce semantics are not a good fit for all problems [9, 37], which may lead to unnaturally expressed solutions and decreased performance [39]. Finally, Google's recent patent on MapReduce [13] may prompt commercial actors to look for an alternative framework.

In this respect, the KPN framework is a viable alternative, but, in practice, very few general-purpose KPN run-time implementations exist. Known examples include the Sesame project [34], the process network framework [1, 27], YAPI [10] and Ptolemy II [5]. PNRunner, a part of the Sesame project, is an event-driven simulator of embedded systems that employs KPNs for application modeling and simulation. However, since an event-driven simulation significantly slows down execution, Sesame is not suitable for executing KPNs where performance is important. The process network framework supports distributed execution and deadlock detection in KPNs, but only a 1:1 scheduling model. It would require substantial extensions to

introduce support for m:n scheduling. YAPI is not a pure KPN implementation, as it extends the semantics and thus introduces the possibility of non-determinism. In addition, it is unclear whether the implementation can use multiple CPUs.[1] Ptolemy II is a Java-based prototyping platform for experimenting with various models of computation, and it spawns one thread for each Kahn process. The amount of code that comprises the JVM would make it prohibitively difficult to experiment with low-level mechanisms, such as context-switches.

Another framework based on the process network paradigm is StreamIt [17]. It is a language and a run-time system for simplifying implementation of stream programs described by a graph consisting of computational blocks (filters) having a single input and output. Filters can be combined in fork-join patterns and loops but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous data-flow process network [25]. The compiler produces code that can exploit multiple machines or CPUs, but their number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

Thus, all of the existing frameworks have some shortcomings that are difficult to address, which has motivated us to implement a new KPN run-time from scratch. Before describing the implementation details of Nornir, we shall first describe the basic KPN semantics in the next section.

3 Kahn process networks

KPNs, MapReduce and Dryad have two important features in common, both of which significantly simplify development of parallel applications: (1) communication and parallelism are explicitly expressed in the application graph; (2) individual processes do not have access to each other's state and can be written in the usual sequential manner. In addition, KPNs have a unique combination of other desirable properties:

- *Determinism.* KPNs are deterministic, i.e., each execution of a network produces the same output given the same input,[2] regardless of scheduling strategy.
- *Reproducible faults.* One consequence of determinism is that faults are consistently reproducible, which is otherwise a notoriously difficult problem with parallel and distributed systems. Reproducibility of faults greatly eases debugging.
- *Composability.* Another consequence of determinism is that processes can be composed: connecting the output of a process computing $f(x)$ to the input of a process computing $g(x)$ is guaranteed to compute $g(f(x))$. Therefore, small KPNs can be developed and tested individually and later put together to perform more complex tasks.
- *Deterministic synchronization.* Synchronization is embodied in the blocking receive operation. Thus, developers need not use other, low-level and non-

[1] Inspection of source code indicates that some support for multiple CPUs is built into YAPI. Experiments on Solaris, however, have revealed that YAPI never actually uses more than a single CPU.

[2] Provided that processes themselves are deterministic.

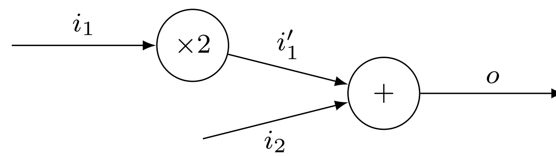


Fig. 1 An example KPN. i_1 and i_2 are external input channels to the network (assumed to be numbers), o is the external output channel, and i'_1 is an internal channel. The inputs and the output are related by the formula $o = 2i_1 + i_2$

deterministic synchronization mechanisms such as mutexes and condition variables.[3]

– *Arbitrary communication graphs.* Whereas MapReduce and Dryad restrict developers to a parallel pipeline structure [37] and directed acyclic graphs (DAGs), KPNs allow *cycles* in the graphs. Because of this, they can directly model iterative algorithms. With MapReduce and Dryad this is only possible by manual iteration, which incurs high setup costs before each iteration [31].
– *No prescribed programming model.* Unlike MapReduce, KPNs do not require that the problem be modeled in terms of processing over key-value pairs. Consequently, transforming a sequential algorithm into a Kahn process often requires minimal modifications to the code, consisting mostly of inserting communication statements at appropriate places.

A KPN [22] has a simple representation in the form of a *directed graph* with *processes* as nodes and *channels* as edges, as exemplified in Fig. 1. A process encapsulates data and a single, sequential control flow, independent of any other process. Processes are only allowed to share data by sending messages over channels. Channels are *infinite* FIFO queues that store discrete *messages*. Channels have *exactly one* sender and *one* receiver process on each end (1:1 communication), and every process can have multiple *input* and *output* channels. Sending a message to the channel always succeeds (from the KPN model point of view), but trying to receive a message from an empty channel *blocks* the process until a message becomes available. Testing for available messages is not permitted. These properties define the *operational semantics* of KPNs and make the Kahn model *deterministic*, i.e., the history of messages produced on the channels does not depend on the process execution order. Every model that relaxes one of these conditions, e.g. allowing non-blocking reads or polling, results in non-deterministic behavior.

The theoretical model of KPNs described so far is idealized in two ways: (1) it places few constraints on process behavior, and (2) it assumes that channels have infinite capacities. These assumptions are somewhat problematic because they allow the construction of KPNs that need unbounded space for their execution. However, any real implementation is constrained to run in finite memory. A common (partial) solution to this is to assign *capacities* to channels and redefine the semantics of send to *block* the sending process if the delivery would cause the channel to exceed its capacity. Under such send semantics, an *artificial deadlock* may occur, i.e., a situation

[3] Inexperienced developers expect that mutexes and condition variables wake up waiting threads in FIFO order, whereas the wake-up order is in reality implementation-dependent and often non-deterministic.

where a cyclically dependent subset of processes blocks on send, although they would continue running in the theoretical model. The algorithm of Geilen and Basten [15] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarges it by one message, thus resolving the deadlock.

It also is worth noting that KPNs are not a universal solution for the inherently difficult problem of developing parallel and distributed applications. Even though determinism is a desirable property from the standpoint of program development, it limits the application areas for KPNs. For example, a disk scheduler is a simple use-case that cannot be appropriately modeled with KPNs. The scheduler must periodically serve all clients in some order, say round-robin, to preserve fairness. However, since read is blocking, absence of requests from one client can indefinitely postpone serving of requests from other clients. Such use-cases mandate use of other frameworks, or relaxing the KPN formalism by introducing non-deterministic construct(s) such as $m:n$ channels and/or polling, which would reduce its conceptual value.

4 Nornir

The Nornir run-time system is implemented in C++, and runs on Windows and POSIX operating systems (Solaris, Linux, etc.). The implementation[4] consists of a Kahn process (KP) scheduler, message transport and deadlock detection and resolution algorithms.

4.1 Process scheduler

Nornir implements both preemptive 1:1 and cooperative m:n scheduling models. In the 1:1 model, one OS thread is used for each KP, blocking mutexes are used to protect the channels, and condition variables are used as the sleep/wake-up mechanism. We have investigated this approach in an earlier paper [38] and showed that the m:n scheduling model is considerably more efficient. Because of this, we focus here on the m:n scheduling model, which we have also used for our performance evaluations.

With an m:n model (see Fig. 2), many KPs are time-multiplexed onto one OS thread, which we call *runner*. Since the m:n scheduler is cooperative, a KP runs uninterrupted until it exits or encounters a blocking point. In our implementation, KPs yield implicitly only in message send and receive operations. However, since KPs may use all UNIX system calls, they can perform I/O in the usual way. In that case, I/O operations may block without causing the calling KP to block or yield. The runner executing the KP will be blocked and unable to dispatch another KP. This can be avoided by using either asynchronous I/O or dedicated runners for KPs that perform I/O.

The m:n scheduler uses user-mode code to switch contexts between a KP and the scheduler core. On Solaris and Linux running on AMD64, we employ optimized, hand-crafted assembly code for context-switch; on other platforms we use OS-provided facilities: fibers on windows, and `swapcontext()` on POSIX. The latter is inefficient because each context-switch requires a system call to save and restore the signal mask.

[4] Code is available at http://simula.no/research/networks/software.

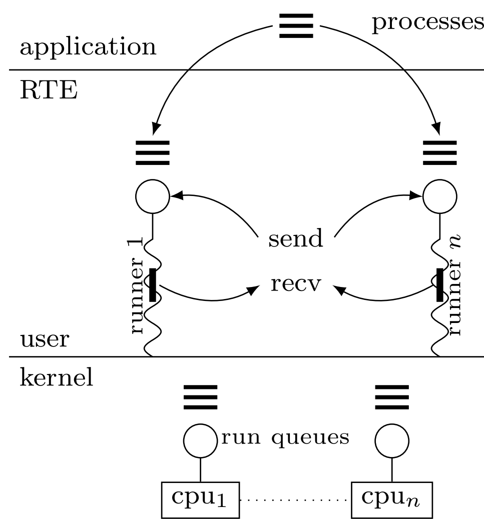


Fig. 2 KP scheduling: each runner has its own private run-queue (*small circles*) containing ready KPs (*small black squares*), and is executing at most one KP at any given time. Runners in Nornir are bound to different CPUs, so they never compete with each other, but they compete with other threads and processes on the machine

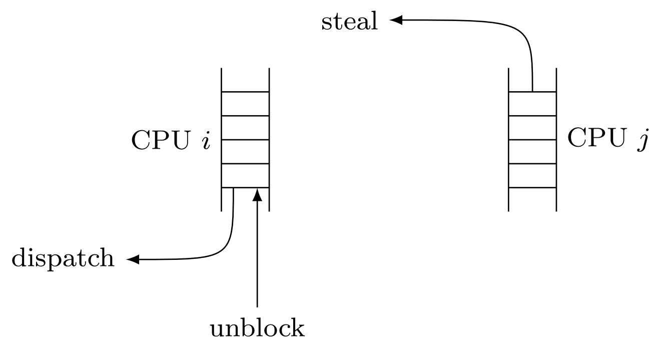


Fig. 3 Queue operations in work-stealing. CPU i accesses its own queue only at the *front*, while it steals tasks from other CPUs only from the *back*

4.2 Scheduling policies

Since KPNs are deterministic, they provide great freedom in implementing the process scheduler: any *fair* scheduler results in a KPN execution that generates the full output.[5] In this context, fairness means that the execution of a ready process will not be indefinitely postponed.

We have thus made Nornir configurable with different scheduling policies. In addition to the classical work-stealing [4] algorithm, we have also implemented and evaluated a modified work-stealing algorithm and a scheduling algorithm based on graph-partitioning [7]. The latter tries to balance the computational load evenly while reducing the amount of inter-CPU synchronization and communication.

The *work-stealing* algorithm [4] is illustrated in Fig. 3. When the KPN is started on a system with m CPUs, m runner threads ("runners") are created and pinned to different CPUs. Each runner has a private run-queue of ready KPs and executes the KP at the front of its queue (*dispatch*). If this queue is empty, it tries to *steal* a KP from the back of the queue of a randomly chosen runner. KPs that become ready (*unblock*) are added at the front of a queue. We use two algorithms for determining this queue. The original algorithm always prepends a newly unblocked KP to the queue of the

[5]If the scheduler is not fair, the output will be correct, but possibly shorter than it would be under a fair scheduler.

CPU that executes that currently running KP which triggered the unblocking (*WS-CUR*). This algorithm causes performance problems for some workloads, including the scatter-gather kind of load. They are caused by high contention over run-queues when stealing. We have therefore modified the work-stealing algorithm so that an unblocked KP is placed on the last CPU that executed it [41] (*WS-LAST*). We do not use the non-blocking queue described in [4]; instead we use an ordinary mutex per run-queue. This not only simplifies the implementation, but is also *required* for implementing our WS-LAST modification since the non-blocking queue does not support all of the operations needed to support the modified algorithm. This might become problematic on machines with many cores, but we deemed that introducing the additional complexity of a non-blocking queue was unnecessary at this stage of our research.

The *graph-partitioning* algorithm assigns weights to the nodes and edges in the process graphs. The weight of a node is proportional to the processing power required by the KP and the weight of an edge is proportional to the communication volume between the two processes incident to that edge. The algorithm tries to partition the graph into disjoint sets of nodes so that all node subsets have approximately equal total node weights. In addition, the algorithm tries to minimize the cut cost, i.e., the total weight of edges crossing partitions. Graph-partitioning is an NP-hard problem, so we have implemented a heuristic based on the work presented by Catalyurek et al. [7].

4.3 Message transport

In KPNs, channels have a twofold role: (1) to interact with the scheduler, i.e., block and unblock processes on either side of the channel, and (2) to transport messages. The initial capacity of the channel may be specified when the channel is first created; if omitted, a default capacity is used.

Interaction with the scheduler is needed particularly for the cases of a full or empty channel. Receiving from an empty channel or sending to a full channel blocks the acting process. Similarly, receiving from a full channel or sending to an empty channel unblocks the process on the other side of the channel.

Message transport over channels is protected by mutexes that are essentially implemented by *busy-waiting*, but yield to the scheduler for every failed locking attempt. Combined with the user-space work-stealing scheduler, this is less wasteful than a spinlock and much faster than a full-fledged sleep/wake-up mechanism. Furthermore, since channels are 1:1, at most two processes compete for access to any given channel, so the expected number of spins in the case of contention on a channel is very small.

Since KPs are executing in a shared address space in our implementation, it is still possible that they modify each other's state[6] and thus ruin the KPN semantics. There are at least two ways of implementing a channel transport mechanism that lessens the possibility of such occurrence:

[6] C++ is an inherently unsafe language, so there is no way of *preventing* this.

- A message can be dynamically allocated and a pointer to it sent in a class that implements *move semantics* (e.g., `auto_ptr` from the C++ standard library).
- A message can be physically copied to/from channel buffers which is, in our case, done by invoking the copy-constructor.

We initially implemented the first approach, which requires dynamic memory (de-)allocation for every message creation and destruction, but is essentially zero-copy. Our current implementation uses the second approach because measurements on Solaris have shown that memory (de-)allocation, despite having been optimized by using Solaris's *umem* allocator, has larger overhead than copying as long as the message size is less than ∼256 bytes. We have not tested sub-allocators separately; the performance would likely be very similar to message copying with a small constant message size.

Since C++ is a statically-typed language, our channels are also *strongly-typed*, i.e., they carry messages of only a single type. Since *communication ports* (endpoints of a channel; used by processes to send and receive messages) and channels are parameterized with the type of message that is being transmitted, compile-time mechanisms prevent sending messages of wrong types. Furthermore, the run-time overhead of dynamic dispatch based on message type is eliminated. Nevertheless, if dynamic typing is desired, it can be implemented by sending byte arrays over channels, or in a more structured and safe manner by using a standard solution such as or C++ with run-time type information or *Boost.Variant* (see http://www.boost.org).

As KPs have only blocking read at their disposal, it is useful to provide an indication when no more messages arrive on the channel (EOF). One way of doing this is to send a message with specific value that indicates EOF. However, all values of a type (e.g., `int`) might be meaningful in a certain context, so no value is available to encode the EOF indication. In such cases, one would be forced to use solutions that are more cumbersome and that impose additional overhead (for example, dynamic memory allocation with `NULL` pointer value representing EOF). We have therefore extended channels by introducing support for *EOF indication*: the sender can set the EOF status on the channel when it has no more messages to send. After EOF on the channel has been set, the receiver is able to read the remaining buffered messages. After all messages have been read, the next call to the port's `recv` method returns false immediately (without changing the target message buffer), and the next `recv` call block the process permanently.

4.4 Deadlock detection and resolution

Deadlock detection and resolution make it possible to execute many[7] KPNs in finite space. Each time a process would block, either on read or on write, a deadlock detection routine is invoked. Since communication is 1:1, read and write are blocking and polling is forbidden, it is straightforward to notice when a ring of blocked processes is closed. The deadlock detection and resolution algorithm in our current implementation uses a centralized data-structure (the blocking graph) and thus must run while

[7] As demonstrated in [37], it is possible to construct KPNs that are under no circumstances executable in finite space.

holding a single global mutex. If no cycle is found, the KP is blocked and this fact is recorded in the blocking graph. Otherwise, the capacity of the smallest channel in the cycle is increased by one, as suggested by [15]. Similarly, when a process receives from a full channel, the upstream KP is unblocked and the blocking graph updated.

4.5 Accounting

We have implemented a detailed accounting system that enables us to monitor many different aspects of Nornir's run-time performance, such as CPU time used by each process, number of context-switches, number of loop iterations in waiting on spin-locks, number of process thefts, number of messages sent to processes on the same or a different CPU. We have measured (see [38] for methodology) that a single transaction, consisting of [send → context switch → receive] operations, takes 1.4 µs with accounting enabled. When accounting is disabled, this time drops to ∼0.68 µs. The largest overhead in our accounting mechanism stems from the measurement of per-KP CPU time, which requires a system call immediately before a KP is run and immediately after a KP returns to scheduler.

5 Performance evaluation

We have evaluated performance aspects of Nornir in particular, but also of the KPN modeling paradigm in general:

- Effects of scheduling policies on performance.
- Scalability of Nornir with respect to KPN size and parallelism granularity.
- Performance consequences of using MapReduce or KPNs for modeling parallel applications.

To investigate these issues, we have designed and implemented four synthetic workloads: an H.264-encoding process graph, an Advanced Encryption Standard (AES) encryption algorithm, a random process network and a pipeline. In addition, we have implemented the canonical MapReduce example, word-frequency program, on top of Nornir in two ways: (1) by implementing the exact MapReduce semantics within the KPN framework and using it to solve the word-frequency problem, and (2) by designing and implementing a KPN that is best suited for the problem. Lastly, we have compared the performance of both of our solutions to the solution implemented on top of Phoenix [31], which is a MapReduce implementation designed specifically for multi-core machines.

In the following subsections, we first describe our experimental setup and methodology and proceed to present results in the order outlined above. Workloads are described in detail as they are encountered.

5.1 Methodology

The test programs have been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). Nornir has been compiled with accounting turned on, since this is necessary to study performance effects of deadlock

detection. We have run them on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM running Linux kernel 2.6.27.3. Each data point is *median*[8] of 9 consecutive measurements of the total real (wall-clock) running time. This metric is most representative because it accurately reflects the real time needed for task completion, which is what the end-users are most interested in.

All benchmarks, except those of Sect. 5.2, have been configured to use our modified work-stealing algorithm and initial channel capacity of 64 messages, with deadlock detection enabled.

5.2 Choice of scheduler

As stated in Sect. 4.1, a KPN yields identical results when run by any fair scheduling algorithm, but the running time may be highly dependent on the chosen algorithm. In this section, we investigate the relative merits of the original work-stealing algorithm (WS-CUR), our modified version (WS-LAST) and the method based on graph-partitioning (GP). In order to separate performance effects of scheduling and deadlock detection, the latter has been turned off for this series of experiments. Nevertheless, all benchmarks did finish without any artificial deadlock problems.

5.2.1 Original vs. modified work-stealing

We have performed extensive comparison of application running times under WS-CUR and WS-LAST policies. We have found that both algorithms lead to very similar application performance,[9] *except* on a specific type of workload where WS-LAST is clearly superior.

The KPN for this workload, which we have named *scatter-gather*, is shown in Fig. 4. This structure is typical for embarrassingly-parallel problems, i.e., problems that can be split into parts that can be processed independently of each other. The AES

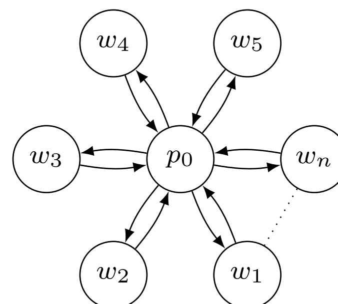


Fig. 4 Scatter-gather topology

[8]We have investigated also correlation with other quantities, such as rate of steal attempts. Using median allowed us to use the values of other measured variables *as is*. Mean value would not correspond to any particular measurement, and it would be unclear how the values of other measured quantities could be interpolated. The observed variation in running times is small, so the difference between using median and mean is negligible.

[9]We have used also the other workloads described later in this paper for our comparisons. On all of our measurements, the difference between mean running times under WS-CUR and WS-LAST was within 2%, most often in favor of WS-LAST.

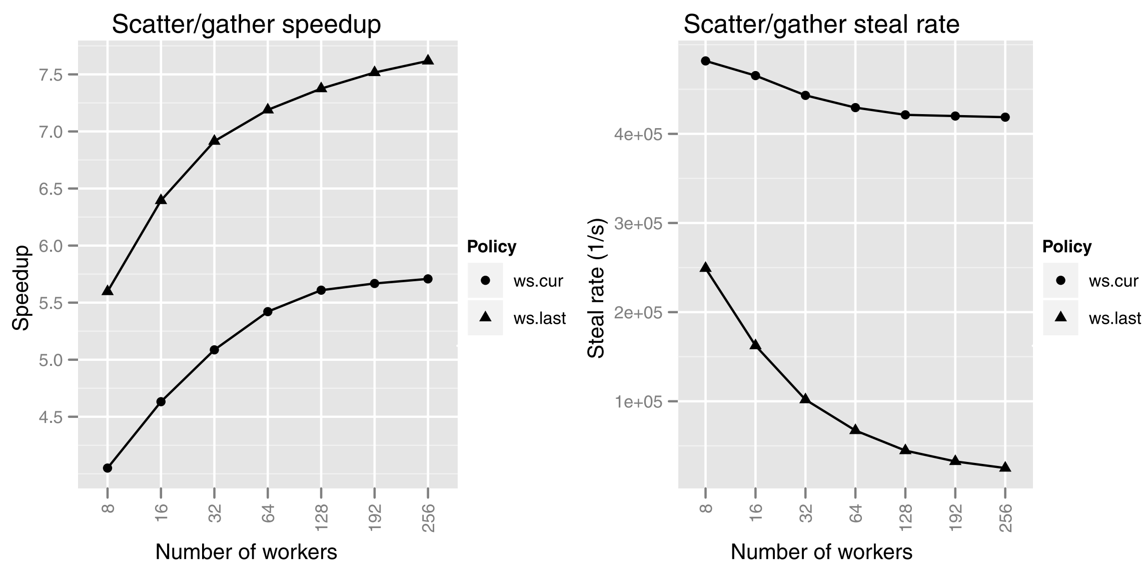


Fig. 5 Scatter-gather on 8 CPUs: speedup and steal rate vs. number of workers

encryption benchmark described later is an example of such a problem. There are also workloads that proceed in stages, where all sub-problems of one stage must be finished before starting work on the next stage; in such cases KP P_0 acts as a barrier. An example of such a situation is the synchronization between Map and Reduce stages of a MapReduce computation. The scatter-gather benchmark executes in m rounds. In each round, P_0 first sends a single message to each of the n workers, and then waits to receive the same number of replies from each worker. The workers are oblivious to the round-based execution: a worker just receives a message, uses a fixed amount of CPU time t and sends a reply to P_0. For the results shown in Fig. 5, we have used $m = 12500$, $t = 10^{-4}$ seconds, and varied n over the set $n \in \{8, 16, 32, 64, 128, 192, 256\}$. Thus, the total useful work performed by all workers is $W = nmt = 1.25n$ CPU seconds.

Compared to running the whole workload on a single core, we can observe that speedup increases with the number of workers. For example, using 256 workers, WS-CUR achieves a speedup of 5.7 whereas WS-LAST achieves a speedup of 7.6. As can be observed from Fig. 5, the speedup is directly related to the reduction in the number of steal attempts per second. The number of steal attempts is reduced because WS-LAST actively distributes the load over all CPUs, thus significantly reducing contention over the run-queues. WS-CUR, on the other hand, always wakes up *all* workers to the *same* CPU, i.e., the one on which P_0 is running. This creates significant contention over a *single* run-queue at the beginning of each round, which leads to significant loss of performance. This contention-reducing property will be even more important for future applications as the number of cores in computer systems is steadily increasing.

5.2.2 *Modified work-stealing vs. graph-partitioning*

Scheduling based on GP needs a mechanism that can detect load-imbalance and trigger repartitioning when the load-imbalance has become intolerably high. For that

purpose, we measure the idle time accumulated by all runners and trigger repartitioning when it exceeds a preset constant threshold τ.

GP has complex performance characteristics which require running an extensive set of benchmarks across different parameter sets. Due to the complexity of accurately characterizing GP behavior, we only list our main conclusions and refer the reader to [37] for full details.

Application running time Compared to GP, WS-LAST yields in general best performance on all workloads. Provided that there is enough parallelism in the program and that a process performs a sufficient amount of work each time it is scheduled, WS-LAST always achieves speedup almost proportional to the number of CPUs. Speedup using GP, however, is always significantly lower—in some pathological cases, programs running under GP on multiple CPUs take *longer* to execute than when running on a single CPU.

To illustrate this, we used the H.264 benchmark[10] that simulates the data flow in an H.264 codec. Figure 6 shows our H.264 KPN, which is only a slight adaptation of the encoder block diagram found in [32]. The functional blocks of the H.264 encoding process are implemented as KPs using synthetic loops to spend the same amount of time for each "encoded" frame that would be used by a real codec. We have obtained these timings by profiling x264, which is an open-source H.264 encoder implementation, and mapping the results to the shown process graph. Since the P, MC and ME stages together use more than 50% of the total CPU time, we have parallelized each of these stages in the same way as shown in Fig. 4. The input video consists of a series of discrete frames, and the encoder operates on small parts of the frame, called macroblocks, typically 16 × 16 pixels in size. The encoder consists of "forward" and "reconstruction" data-paths which meet at the prediction block (P). The

Fig. 6 H.264 block-diagram, adapted from the H.264 white paper [32] where the inputs to the codec are the current (F_n) and reference (F_{n-1}) frames. Here, each *square block* assumes the role of P_0 in Fig. 4 and has attached n workers

[10]This example demonstrates that feedback loops are not only a matter of convenience, but actually *essential*. The H.264 benchmark is a synthetic workload for the expressive power of a programming framework. Thus, as already argued in the introduction, neither MapReduce nor Dryad can be used to implement the H.264 encoder.

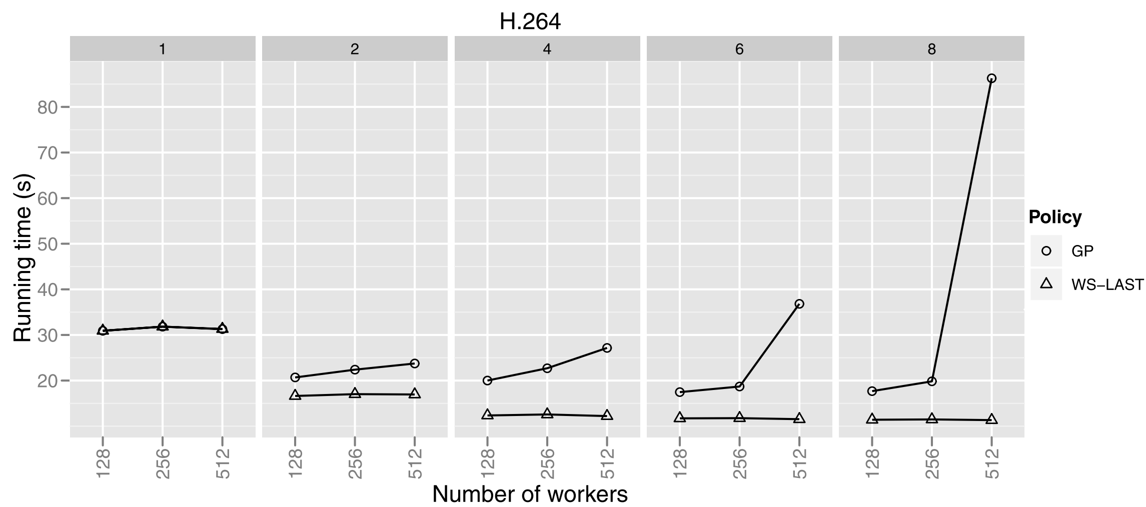


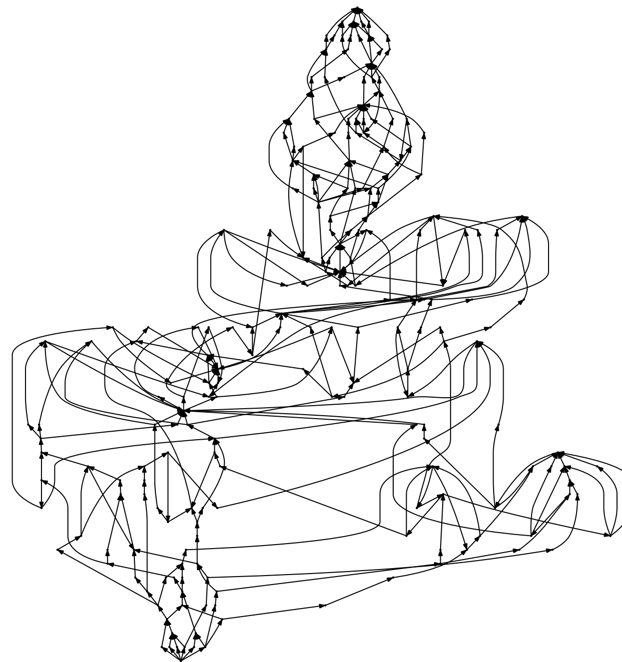
Fig. 7 Median running times for WS-LAST and GP on 1, 2, 4, 6 and 8 CPUs

role of the prediction block is to decide whether the macroblock is encoded by using intra-prediction (relative to macroblocks in the *current* frame F_n) or inter-prediction (relative to macroblocks in the *reference* frame(s) F_{n-1}). The encoded macroblock goes through the forward path and ends at the entropy coder (EC), which is the final output of the encoder. The decision on whether to apply intra- or inter-prediction is based on factors such as the desired bandwidth and quality. To be able to estimate quality, the codec needs to apply transformations inverse to those of the forward path, and determine whether the *decoded* frame satisfies the quality constraints.

In Fig. 7, we plot the performance results for both GP and WS-LAST running the H.264 benchmark. The main reason of performance degradation with increasing number of workers is the limited parallelism available in the H.264 network. Under GP, runners accumulate idle time faster than our simple heuristics for deciding when to rebalance the load is able to catch up. This means frequent repartitioning, which is protected by a single mutex, and process migration. These operations are performed hundreds of times per second, leading to severe performance degradation.

Variance of application running times Our experiments have confirmed that GP indeed fulfills its design goal of reducing traffic between processes on different CPUs. However, we have also observed that its load-balancing properties are rather poor. This is illustrated using a random network that is a directed graph consisting of a source KP, a number of intermediate KPs arranged in n_l layers (user specified) and a sink KP (see Fig. 8 for an example). The number of KPs in each layer is randomly selected between 1 and the user-specified maximum number m. The intention of this construction is to mimic, with fine granularity, network protocol graphs or parallelism in algorithms. The network may have additional back-edges that create cycles, with each node being constrained to have at most one back-edge, be it outgoing or incoming. The workload is generated by sending and receiving messages that contain plain integers; each integer denotes the amount of CPU time that the receiving KP spends on processing, using a synthetic loop, before sending a reply or receiving another message. More specifically, the source KP sends n messages, each containing a fixed amount of work corresponding to w seconds of CPU time. A KP reads a single message from each of its n_i inputs and adds them together to compute the total amount

Fig. 8 A random graph with 212 nodes, 333 edges and 13 cycles with the source node on *bottom* and sink node on *top*



t of CPU-time that is to be consumed. Then, the KP runs a synthetic loop to use the number of CPU seconds proportional with t, and then it distributes t to its n_o forward out-edges. A message representing $\lfloor t/n_o \rfloor$ seconds of CPU time is sent to each of the n_o forward out-edges; if t is not divisible by n_o then the remainder is added to the last outgoing edge. Lastly, if a KP has a back-edge, a message is sent/received (depending on the edge direction) along that channel. As such, the workload w in a single message sent from the source KP equals the workload w collected by the sink KP. Messages sent along back-edges do not contribute to the network's workload; their purpose is solely to generate more complex synchronization patterns. The total CPU load W generated by the network is determined by the formula $W = nT/d$, with each single message representing a CPU load of $w = T/d$ CPU seconds. Here, T is the number of iterations of our synthetic loop that uses 1 second of CPU time on the benchmark machine, and d is a quantity that we have named *work division factor*. For example, when $d = 10{,}000$, each message by the source represents 10^{-4} seconds (100 µs) of CPU time. This allows us to control the granularity of chunks into which the total workload W is divided.[11]

Figure 9 shows a box-and-whiskers plot of the distribution of running times and number of repartitionings for different values of the τ parameter; $\tau = 0$ corresponds to WS-LAST. As expected, larger τ causes fewer repartitionings. However, it has very little influence on the running time. When $\tau \geq 72$, the *minimal* measured running times are approximately constant, and around ~5 s (WS-LAST achieves running time of ~2.5 s), while the *maximal* running times vary widely. In general, the *variance* of running times for any given τ is large, i.e., the running time of any given experiment run is very unpredictable and uncorrelated with the repartitioning frequency. We

[11]Note that T/d equals the *maximum* work performed by any single KP. The *lower* bound on the amount of work performed by any KP is T/md where m is the maximal number of KPs in any layer.

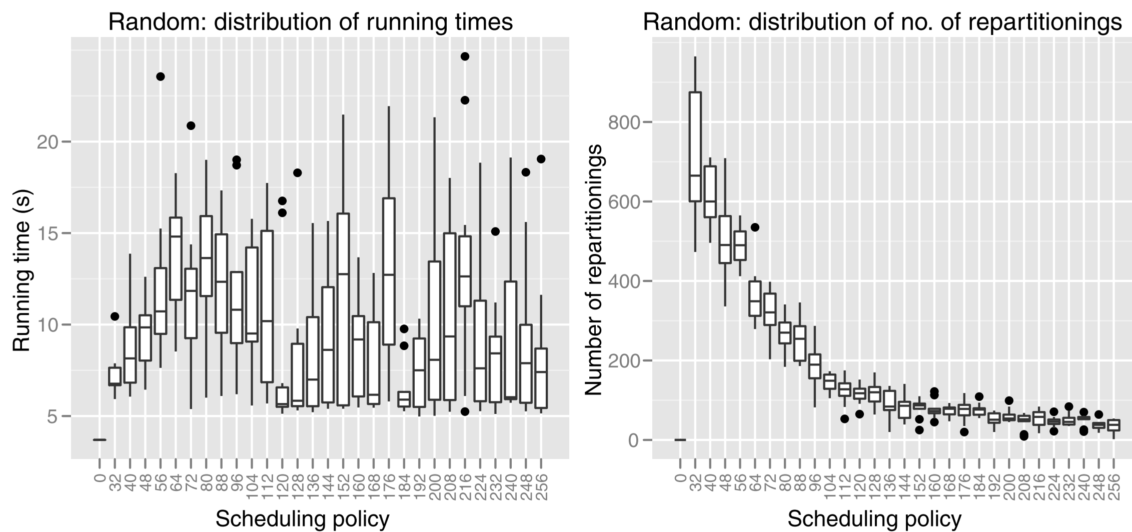


Fig. 9 Illustration of GP variance in running time for the Random workload on 8 CPUs and $d = 1000$, which is one step past the WS peak speedup. x-axis is the value of the idle time parameter τ for the GP policy, and $\tau = 0$ shows the results for WS

deem that this is the biggest drawback of GP-based scheduling and therefore recommend that users perform extensive performance tests with their particular workloads before employing GP-based scheduling in production.

Traffic patterns As explained earlier, the GP algorithm is designed to satisfy two mutually incompatible constraints: balance the load across multiple CPUs and minimize the traffic between processes on different CPUs. We have observed that there is much more local traffic under GP scheduling. For example, on the random graph benchmark under GP on 8 CPUs, *at least* 70% of all messages are communicated between KPs on the same CPU for all values of work division factor. On the same benchmark under WS-LAST, *at most* 20% of all messages constitute local traffic, and this ratio decreases as work division increases.

Based on these results, we use WS-LAST for the rest of the experiments evaluating different application scenarios. Note, however, that we are running on a single, multicore machine. When the communication cost increases, for example by extending the investigation to a compute cluster, the benefit of using graph-partitioning will increase (but this is subject to further experiments).

5.3 Scalability of Nornir

We have used several different benchmarks to evaluate scalability of Nornir as well as the overheads of centralized deadlock detection. To this end, we present results from H.264, AES, random graph and pipeline benchmarks. We are generally interested in speedup over one CPU and in factors that cause the speedup to be less than the ideal.

5.3.1 H.264

In the first experiment, we have used the artificial H.264 encoding workload described in Sect. 5.2.2 (see block diagram in Fig. 6). We have configured the benchmark to encode 30 video frames at rate of 1 frame per second (fps), with the number of workers

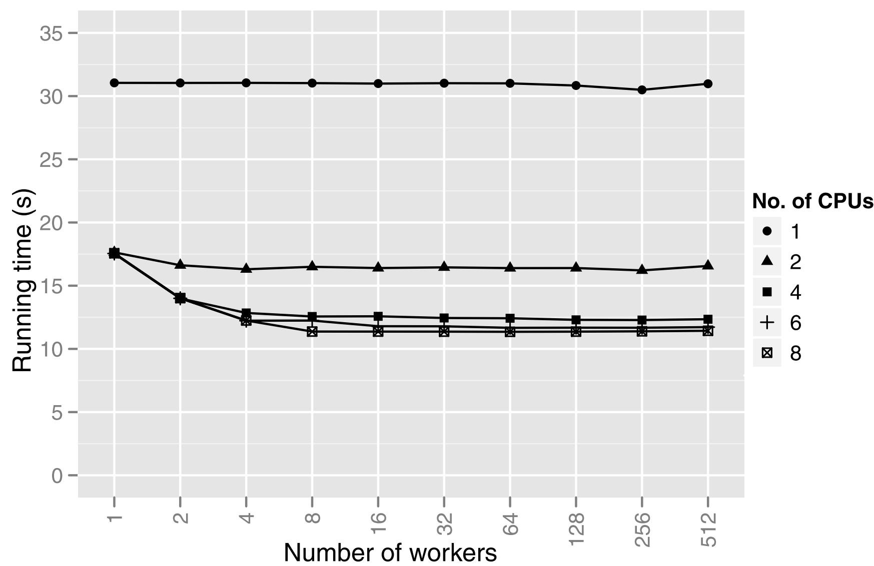


Fig. 10 H.264: performance of "encoding" 30 frames at ∼1 fps

varying from 1 to 512 in steps of powers of two. From the results in Fig. 10, we can observe that the performance gets slightly better as the number of workers per stage increases up to the number of CPUs, and remains constant afterwards. The best achieved speedup on 8 CPUs is only ∼2.8; this limitation is caused by data dependencies in the algorithm.

5.3.2 AES

The AES encryption is performed in ECB mode, resulting in a typical example of an embarrassingly parallel workload, i.e., a task that can be divided into subtasks that can execute completely independently of each other.[12] The AES KPN is the same as the KPN shown in Fig. 4; this topology is in fact common for all embarrassingly parallel algorithms. The source P_0 hands out equally-sized memory chunks to n workers where the exchanged messages are carrying only pointer-length pairs (16 bytes in total). When a worker receives a message, it encrypts the chunk in-place with the 128-bit AES algorithm using several passes and sends the reply message back to the source KP.

In this benchmark, we have set the total block size to 2^{28} bytes (256 MB), and the total number of workers has been varied from 1 to 2048. P_0 (refer to Fig. 4) partitions the memory block into as many non-overlapping chunks as there are workers and sends each chunk to a worker for encryption. The number of encryption passes has been set to 40 so that the total running time is sufficiently high. The results, shown in Fig. 11, show perfect linear speedup with the number of CPUs (running times of 80 and 10 s on 1 and 8 CPUs), as soon as the number of workers becomes greater or equal to the number of CPUs. When $w < 8$ the speedup is limited only by the lack of work to be assigned to all available CPUs; in particular, for $w = 1$, the speedup is 1 because there is only a single worker process encrypting the whole block.

[12]ECB mode should never be used in practice; CTR mode is far more secure and also yields embarrassingly parallel workload.

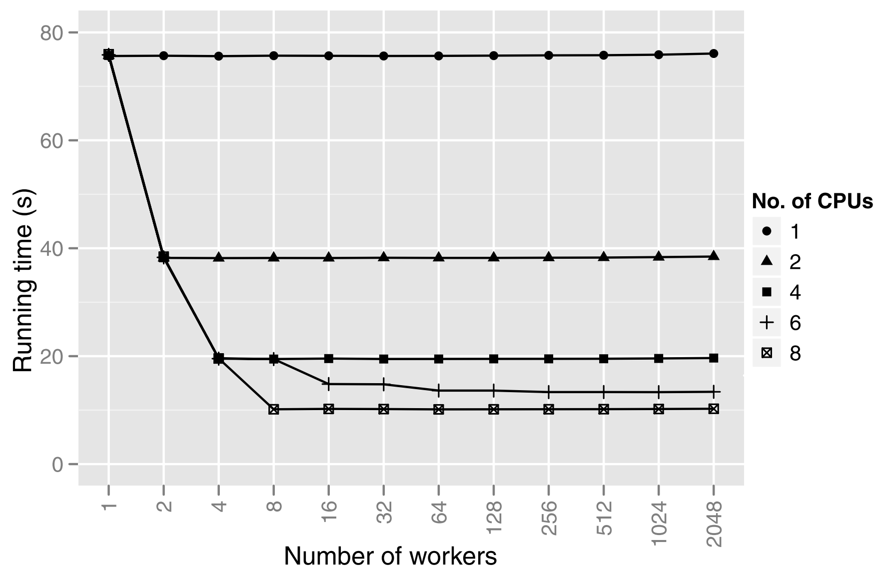


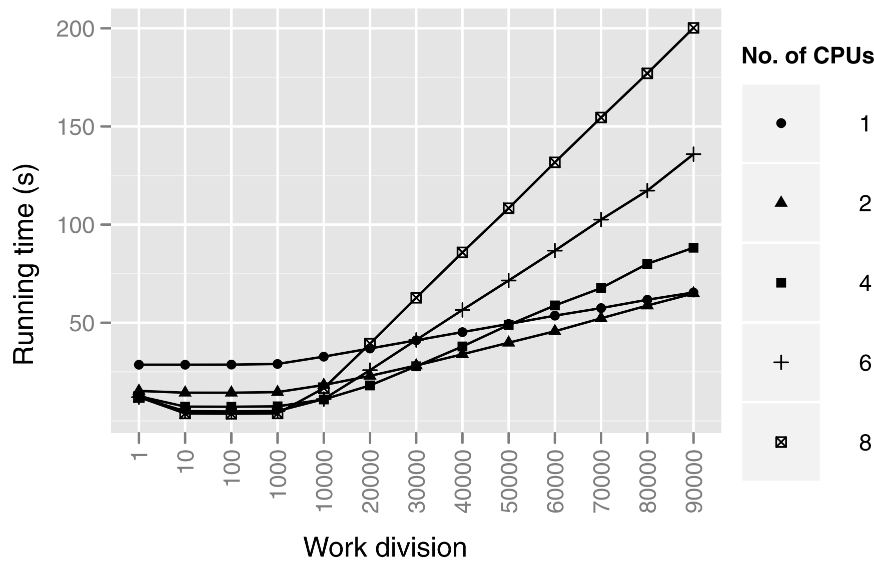
Fig. 11 AES encryption benchmark

5.3.3 Random network

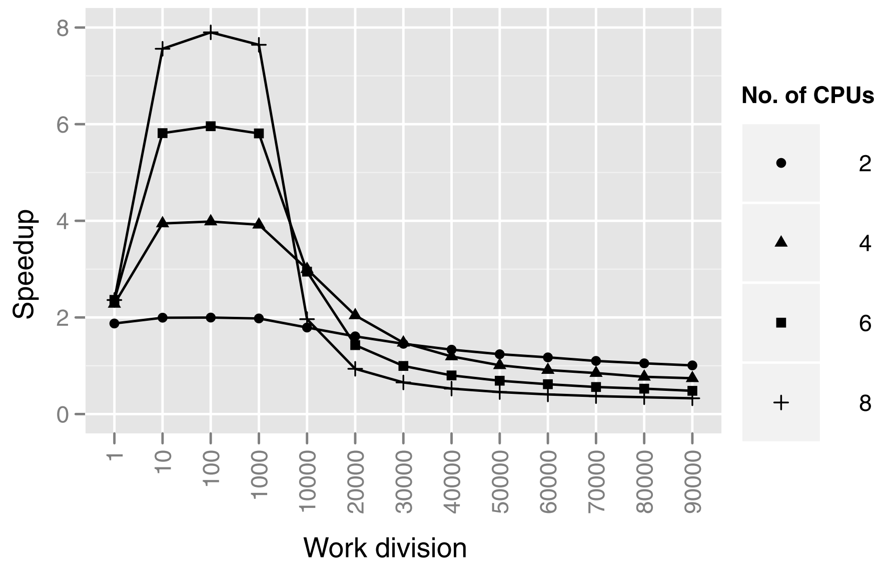
The scalability of Nornir using the Random benchmark described in Sect. 5.2.2 is shown in Fig. 12(a). This plot shows the absolute running times of a random graph with cycles at different values of d, whereas Fig. 12(b) shows speedup over running time on 1~CPU. In our experiments, we have set $n = d$. As d increases to 100, the available parallelism in the network increases, and the running time decreases; in Fig. 12(b) we see that $d = 100$ achieves perfect speedup on multiple CPUs. At $d = 1000$, running time starts increasing linearly with d, and grows faster on more CPUs. This is caused by frequent deadlock detections, as witnessed by Fig. 13, which shows the number of started deadlock detections per second of running time. Since deadlock detection uses a global lock to protect the blocking graph, this limits Nornir's scalability on this benchmark. A possible way of avoiding this problem, in our current implementation, is to increase default channel capacity to a larger value. A long-term solution is implementing a distributed deadlock detection and resolution algorithm [1].

5.3.4 Pipeline

A pipeline does the same kind of processing as the random network, except that each layer has exactly one process and each process takes its only input from the preceding process in the pipeline. In all previous benchmarks, the communicated messages have been rather small (less than 32 bytes). We have used a pipeline consisting of 50 stages to study the impact of message size on performance. As previously, d messages have been generated by the source process, each containing $1/d$ seconds of CPU time. A noticeable slow-down (see Fig. 14) happens regardless of message size and only at $d = 10^5$, which is equivalent to 10 µs of work per message, which is only 7 times greater than the time needed for a single transaction (defined in Sect. 4.5). The drop in performance is proportional to message size and the number of CPUs. Copying larger



(a) Running time.



(b) Speedup over 1 CPU.

Fig. 12 Benchmark results of a the random directed graph shown in Fig. 8 in 50 layers. The x-axis is not uniform

messages consumes more time and causes greater contention over channel locks with increasing number of CPUs.

5.4 Comparison with MapReduce

KPNs are generally more flexible than MapReduce in that more general process graphs, e.g., with branches and cycles, can be processed. However, an interesting issue is how these two models perform for applications they both can execute. To evaluate whether the KPN model has any performance advantages over the MapReduce model, we have therefore implemented the word count application, which is the canonical MapReduce example (more examples are given in [37]). We used Nornir to process a MapReduce-version of Word Count (KPN-MR) by constructing a MapRe-

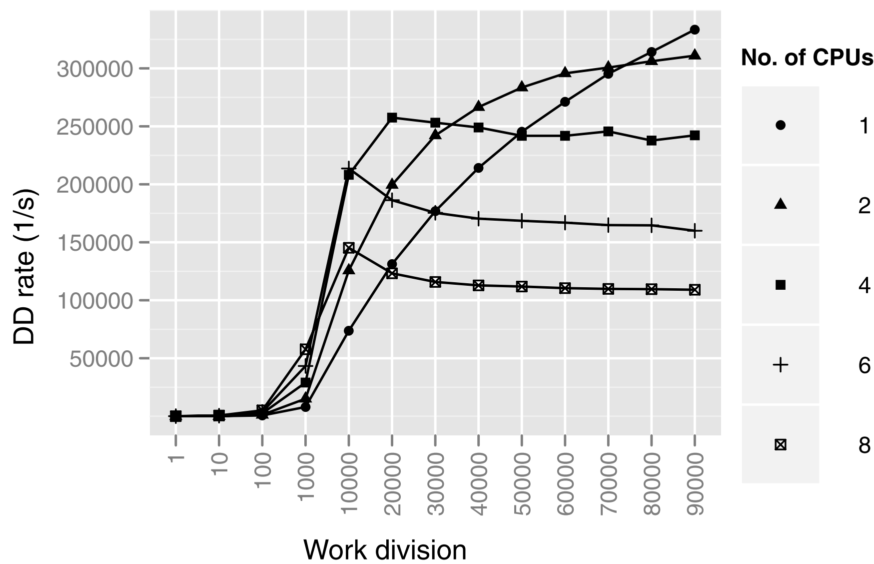


Fig. 13 Deadlock detection rate on the random graph benchmark

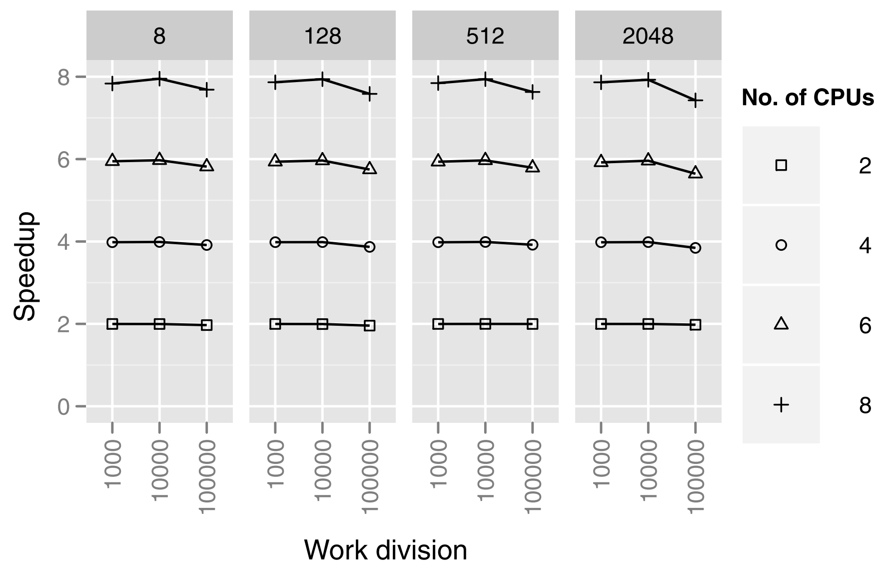


Fig. 14 Pipeline speedup for message sizes of 8, ..., 2048 bytes and three different work divisions (d)

duce topology. In Fig. 15, KPN-MR is compared with Nornir and its more flexible process topology tailored to the problem (KPN-FLEX). We processed a data set that is distributed with Phoenix and used for Phoenix MapReduce benchmarks [31], where the files had the sizes 10, 50 and 100 MB. We can observe that the running time decreases as more CPUs are used and that the KPN-FLEX version has several *times* better performance than the KPN-MR version. Our results indicate that both the KPN-MR and KPN-FLEX implementations scale approximately logarithmically with the number of CPUs: the program running time decreases approximately linearly with each doubling of the number of runners. We believe that the speedup is sub-linear because of two factors: algorithmic complexity of sorting, which is $O(n \log n)$, and the effects of deadlock detection. We also observe that KPN-FLEX is consistently faster than KPN-MR, by a factor of 3–6.7, and the speedup is proportional with the problem size.

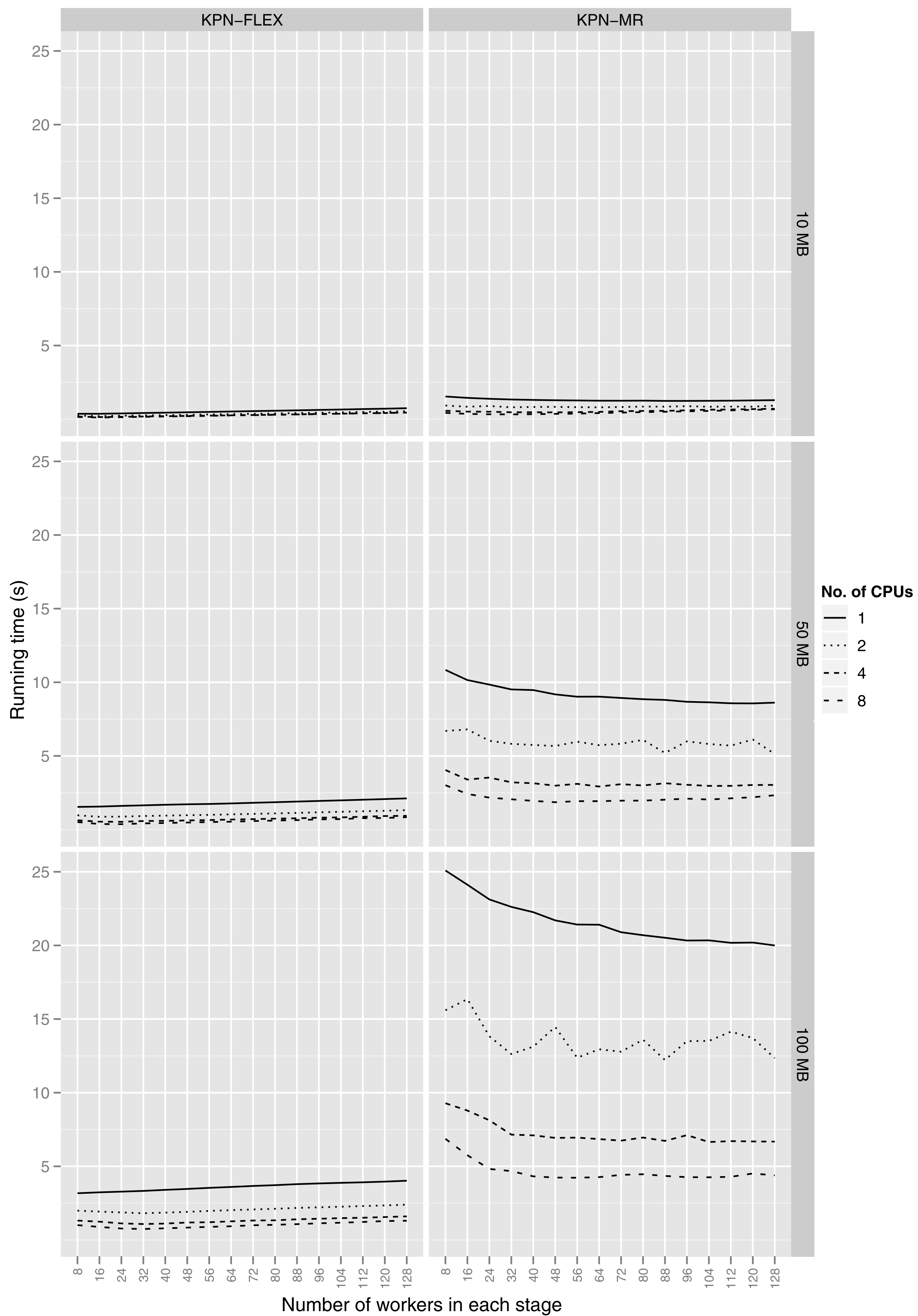


Fig. 15 Running times of the word frequency program on 10, 50 and 100 MB data sets, using KPN-FLEX and KPN-MR solutions

Table 1 Mean running times in seconds (t) of the word frequency programs for Phoenix, KPN-MR and KPN-FLEX. *Speedup* is the speedup factor over Phoenix

Size	Phoenix	KPN-MR		KPN-FLEX	
	t	t	*Speedup*	t	*Speedup*
10	0.30	0.32	0.94	0.11	2.73
50	0.98	1.86	0.53	0.37	2.65
100	2.04	4.23	0.48	0.74	2.76

We have also compared the performance of our KPN-FLEX and KPN-MR implementations with Phoenix [31], which is a MapReduce implementation designed specifically for multi-core machines. We have run Phoenix with its default settings, which means that it uses as many threads in each of the Map and Reduce stages as there are CPUs on the machine. Table 1 compares the Word Count results of Phoenix, KPN-MR and KPN-FLEX. The KPN-MR program is consistently slower than Phoenix, by a factor of 1.06 on the 10 MB file, and by a factor of about 2 on 50 MB and 100 MB files. This result is not very surprising since Phoenix is optimized for running MapReduce programs, while our KPN run-time includes supports for arbitrary directed graphs of communicating processes. However, the KPN-FLEX program, by having the ability to use data structures that are more suited to the given task (hash tables) and avoiding unnecessary work that MapReduce semantics requires (extra sort), achieves about 2.7 times better performance than Phoenix.

5.5 Summary

We have evaluated several aspects of Nornir: scalability of the scheduler with the number of processes and CPUs, overhead of message-copying and overhead of centralized deadlock-detection and resolution. Our findings can be summarized as follows:

- Nornir can efficiently handle a large number of processes. Indeed, in the embarrassingly parallel AES benchmark, it achieved an almost perfect linear speedup of 7.5 on 8 CPUs with 2048 processes.
- Message sizes up to 512 bytes have negligible impact on performance. The cost of message copying starts to be noticeable at message sizes of 2048 bytes. Protecting channels with mutexes has negligible performance impact on 8 CPUs.
- As shown by the pipeline benchmark, context-switch and message-passing overhead starts to have a noticeable impact on the overall performance when the amount of work per message is less than ∼7 times the transaction time (see Sect. 4.5).
- The centralized deadlock detection and resolution algorithm can cause serious scalability and performance problems for certain classes of applications. In our evaluation, this was the case *only* for the random graph benchmark.
- Again, as shown by the random graph benchmark, the default channel capacity of 64 bytes, which we have used in our benchmark, can be too small in certain cases. Increasing it would mitigate overheads of deadlock detection, but it would also increase memory consumption.

- Performance can be further increased by turning off detailed accounting in cases where it is not needed.
- We have not noticed any scalability problems using mutexes for protecting the scheduler's queues instead of using non-blocking queues proposed by Arora et al. [4], except in the scatter-gather benchmark. There, our WS-LAST variant largely mitigates the problem.
- Compared to MapReduce models, Nornir using KPNs is more flexible, and outperforms both applications running in Nornir following the MapReduce model and the Phoenix implementation of MapReduce [31] for multi-core machines.

Although there is room for improvement in Nornir (especially in deadlock detection), our results indicate that message-passing and KPNs in particular are a viable programming model for high-performance parallel applications on shared-memory architectures.

6 Conclusion and future work

In this paper, we have described the implementation details of Nornir, our run-time environment for executing parallel applications specified in the high-level framework of Kahn process networks. KPNs allow branches and cycles in the communication graph of the program, a feature crucial for implementing iterative algorithms such as H.264 encoding. Nornir thus *complements* existing frameworks such as MapReduce and Dryad. We have shown that even for problems without branches and cycles, the flexibility of KPNs makes it possible to design solutions that outperform the solutions cast into the MapReduce framework.

We have evaluated Nornir's efficiency with several synthetic applications (H.264 encoding, random KPN, pipeline, AES) on an 8-core machine. Our results indicate that Nornir can scale well, but that in certain cases (random KPN) the centralized deadlock detection is detrimental for performance, and that a default channel capacity of 64 bytes is too small for some applications. We have also experienced that copying semantics of message-passing have a slight but noticeable impact on performance at message sizes of ∼2048 bytes. Furthermore, Nornir can support parallelism at fine granularity: its overhead becomes noticeable at processing time of 10 µs per message, which is ∼7 times greater than the combined overhead of scheduling and message-passing.

Our study of scheduling policies has led to two results that are significant also outside the context of KPNs. First, we have developed a simple improvement of the work-stealing algorithm that performs as well as the original algorithm but does not exhibit pathologically bad performance with scatter-gather types of workloads. Second, we were the first to evaluate performance of unstructured workloads on multi-core machines when scheduled by a scheduler based on graph-partitioning. We have shown that graph-partitioning not only performs worse than work-stealing but also has very unpredictable running times, an aspect not discussed by Catalyurek et al. [7]. The latter finding is relevant also in the context of resource provisioning in distributed systems where the same types of algorithms are used.

The first, and most important, step in our future work is increasing Nornir's scalability by replacing a centralized deadlock detection algorithm with a distributed one [1]. This will also be the first step towards a distributed version of Nornir, executing on a cluster of machines. For building a distributed version of Nornir, we plan to use MPI to implement message-passing between KPs running on different machines. Further performance increases can be gained by using non-blocking data structures. In the scheduler, we might need to use the non-blocking queue presented in [4] instead of mutexes in order to support scalability beyond 8 CPUs. We might also use a single-producer, single-consumer FIFO queue [16] to avoid yielding between enqueue and dequeue attempts. Since yielding implies switching to a new control flow and a new stack, we expect that this improvement will further increase performance by reducing pressure on data and instruction caches.

Acknowledgements This work has been performed in the context of the *iAD* (Information Access Disruptions) centre for Research-based Innovation, project number 174867.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Allen G, Zucknick P, Evans B (2007) A distributed deadlock detection and resolution algorithm for process networks. In: IEEE international conference on acoustics, speech and signal processing, (ICASSP) 2, April 2007, pp II-33–II-36
2. Apache Hadoop, Accessed July 2009. http://hadoop.apache.org/
3. Armstrong J (2007) A history of Erlang. In: HOPL III: Proceedings of the 3rd ACM SIGPLAN conference on history of programming languages, pp 6-1–6-26. ACM, New York
4. Arora NS, Blumofe RD, Plaxton CG (1998) Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of ACM symposium on parallel algorithms and architectures (SPAA). ACM, New York, pp 119–129
5. Brooks C, Lee EA, Liu X, Neuendorffer S, Zhao Y, Zheng H (2008) Heterogeneous concurrent modeling and design in Java (vol 1: Introduction to Ptolemy II). Tech rep UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008
6. Buhr PA, Stroobosscher RA (1990) The μ system: providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. Softw Pract Exp 20(9):929–964
7. Catalyurek U, Boman E, Devine K, Bozdag D, Heaphy R, Riesen L (2007) Hypergraph-based dynamic load balancing for adaptive scientific computations. In: Proc of 21st international parallel and distributed processing symposium (IPDPS'07). IEEE Press, New York. Also available as Sandia National Labs Tech Report SAND2006-6450C
8. Chaiken R, Jenkins B, Larson P-Å, Ramsey B, Shakib D, Weaver S, Zhou J (2008) Scope: easy and efficient parallel processing of massive data sets. Proc VLDB Endow 1(2):1265–1276
9. Chih Yang H, Dasdan A, Hsiao R-L, Parker DS (2007) Map-Reduce-Merge: simplified relational data processing on large clusters. In: Proceedings of ACM international conference on management of data (SIGMOD), pp 1029–1040
10. de Kock E, Essink G, Smits WJM, van der Wolf R, Brunei J-Y, Kruijtzer W, Lieverse P, Vissers KA, Yapi K (2000) Application modeling for signal processing systems. In: Proceedings of design automation conference, pp 402–405
11. de Kruijf M, Sankaralingam K (2007) MapReduce for the Cell BE architecture. University of Wisconsin Computer Sciences technical report CS-TR-2007 1625
12. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of symposium on operating systems design & implementation (OSDI). USENIX Association, Berkeley, p 10

13. Dean J, Ghemawat S (2010) System and method for efficient large-scale data processing. US Patent No 7650331, Jan 2010
14. Gedik B, Andrade H, Wu K-L, Yu PS, Doo M (2008) Spade: the system s declarative stream processing engine. In: SIGMOD '08: proceedings of the 2008 ACM SIGMOD international conference on management of data. ACM, New York, pp 1123–1134
15. Geilen M, Basten T (2003) Requirements on the execution of Kahn process networks. In: Programming languages and systems, European symposium on programming (ESOP). Springer, Berlin, pp 319–334
16. Giacomoni J, Moseley T, Vachharajani M (2008) FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: PPoPP: proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 43–52
17. Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS-XII: proceedings of the 12th international conference on architectural support for programming languages and operating systems. ACM, New York, pp 151–162
18. He B, Fang W, Luo Q, Govindaraju NK, Wang T (2008) Mars: a MapReduce framework on graphics processors. In: PACT '08: proceedings of the 17th international conference on parallel architectures and compilation techniques. ACM, New York, pp 260–269
19. Hudak P, Hughes J, Jones SP, Wadler P (2007) A history of Haskell: being lazy with class. In: HOPL III: proceedings of the 3rd ACM SIGPLAN conference on history of programming languages, pp 12-1–12-55. ACM, New York
20. Intel Corporation, Threading building blocks. http://www.threadingbuildingblocks.org
21. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) Dryad: distributed data-parallel programs from sequential building blocks. In: Proc of the ACM SIGOPS/EuroSys European conference on computer systems. ACM, New York, pp 59–72
22. Kahn G (1974) The semantics of a simple language for parallel programming. Inf Process 74
23. Knuth DE (1997) Fundamental Algorithms. The Art of Computer Programming, vol 1. Addison–Wesley, Reading
24. Lämmel R (2007) Google's MapReduce programming model—revisited. Sci Comput Program 68(3):208–237
25. Lee EA, Parks T (1995) Dataflow process networks. Proc IEEE 83(5):773–801
26. Message passing interface forum, Accessed July 2009. http://www.mpi-forum.org/
27. Olson A, Evans B (2005) Deadlock detection for distributed process networks. In: ICASSP: Proc of IEEE international conference on acoustics, speech, and signal processing, March 2005, vol 5, pp 73–76
28. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In: SIGMOD '08: proceedings of the 2008 ACM SIGMOD international conference on management of data. ACM, New York, pp 1099–1110
29. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: parallel analysis with Sawzall. Sci Program 13(4):277–298
30. PVM (Parallel Virtual Machine), Accessed August 2010. http://www.csm.ornl.gov/pvm/
31. Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C (2007) Evaluating MapReduce for multi-core and multiprocessor systems. In: Proceedings of the IEEE international symposium on high performance computer architecture (HPCA). IEEE Computer Society, Washington, pp 13–24
32. Richardson IEG H.264/MPEG-4 part 10 white paper. Available online. http://www.vcodex.com/files/h264_overview_orig.pdf
33. The OpenMP API specification for parallel programming, Accessed July 2009. http://openmp.org/wp/
34. Thompson M, Pimentel A (2007) Towards multi-application workload modeling in sesame for system-level design space exploration. In: Embedded computer systems: architectures, modeling, and simulation, vol 4599/2007, pp 222–232
35. Valvåg SV, Johansen D (2008) Oivos: Simple and efficient distributed data processing. In: Proceedings of IEEE international conference on high performance computing and communications (HPCC), pp 113–122
36. Valvåg SV, Johansen D (2009) Cogset: A unified engine for reliable storage and parallel processing. In: Proceedings of IFIP international conference on network and parallel computing workshops (NPC), pp 174–181
37. Vrba Ž (2009) Implementation and performance aspects of Kahn process networks. PhD thesis, Department of Informatics, University of Oslo, Norway, Dec 2009. Dissertation No 903

38. Vrba Ž, Halvorsen P, Griwodz C (2009) Evaluating the run-time performance of Kahn process network implementation techniques on shared-memory multiprocessors. In: International conference on complex, intelligent and software intensive systems (CISIS)—international workshop on multi-core computing systems (MuCoCoS), pp 639–644
39. Vrba Ž, Halvorsen P, Griwodz C, Beskow P (2009) Kahn process networks are a flexible alternative to MapReduce. In: IEEE international conference on high performance computing and communications (HPCC), pp 154–162
40. Vrba Ž, Halvorsen P, Griwodz C, Beskow P, Johansen D (2009) The Nornir run-time system for parallel programs using Kahn process networks. In: 6th international conference on network and parallel computing (NPC), October 2009. IEEE Computer Society, Los Alamitos, pp 1–8
41. Vrba Ž, Halvorsen P, Griwodz C (2010) A simple improvement of the work-stealing scheduling algorithm. In: International conference on complex, intelligent and software intensive systems (CISIS)—international workshop on multi-core computing systems (MuCoCoS), pp 925–930

Paper IX

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Paul B. Beskow, Håvard Espeland, Håkon K. Stensland, Preben N. Olsen, Ståle Kristoffersen, Espen A. Kristiansen, Carsten Griwodzand Pål Halvorsen

Published: Poster, EuroSys 2011

Author contribution: Beskow and Espeland were the primary contributors and investigators of this work, where Beskow made the poster and both presented the work during the poster session.

[simula . research laboratory]

UNIVERSITY OF OSLO

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Paul B. Beskow, Håvard Espeland, Håkon K. Stensland, Preben N. Olsen, Ståle Kristoffersen
Espen A. Kristiansen, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway and IFI, University of Oslo, Norway

P2G is a framework designed to integrate concepts from modern batch processing frameworks into the world of real-time multimedia processing, where we seek to scale transparently with the available resources. P2G consists of a compiler and run-time that analyzes dependencies dynamically and merges or splits kernel instances based on resouce availability and performance monitoring.

KERNEL LANGUAGE

fetch <field F> (age A) [index X] ... [index Z]

Extract a slice of data from a field

store <field F> (age A) [index X] ... [index Z]

Store a slice of data to a field

P2G Kernel language Example

Field definitions:

```
0  int32[] m_data age;
1  int32[] p_data age;
```

Kernel definitions:

```
0  init:
1    local int32[] values;
2
3    %{
4     int i = 0;
5     for( ;i < 5; ++i )
6     {
7      put( values, i+10, i );
8     }
9    %}
10
11   store m_data(0) = values;
12
13
```

```
0  mul2:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = m_data(a)[x]
6
7    %{
8     value *= 2;
9    %}
10
11   store p_data(a)[x] = value;
12
13
```

```
0  plus5:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = p_data(a)[x]
6
7    %{
8     value += 5;
9    %}
10
11   store m_data(a+1)[x] = value;
12
13
```

```
0  print:
1    age a;
2    local int32[] p, m;
3
4    fetch p = p_data(a)
5    fetch m = m_data(a)
6
7    %{
8     for(int i=0; i < extent(p, 0))
9     {
10     cout << "p: " << get(p, i);
11     cout << "m: " << get(m, i);
12    }
13   %}
```

C++ Equivalent

```
void print( int * data, int num )
{
  for( int i = 0; i < num; ++i)
    std::cout << data[i] << " ";
  std::cout << std::endl;
}

int main()
{
  int data[] = { 10, 11, 12, 13, 14 };
  int num = (sizeof data/ sizeof *data);
  print( data, num );
  while(true)
    {
      for( int i = 0; i < num; ++i)
        {
          data[i] *= 2;
        }
      print( data, num );
      for( int i = 0; i < num; ++i)
        {
          data[i] += 5;
        }
      print( data, num );
    }
  return 0;
}
```

P2G ARCHITECTURE

Implicit static dependency graph (IS-DG)

Receive workload

Master node

Refine graph

Partition graph

Refined implicit static dependency graph (RIS-DG)

Distribute full/partial workload to execution nodes

Execution node

Execute workload

Instrumentation data

Dynamically created directed acyclic graph (DC-DAG)

K-MEANS CLUSTERING

Workload description

K-means clustering is an iterative algorithm for cluster analysis, which aims to partition *n* datapoints into *k* clusters in which each datapoint belongs to the cluster with the nearest mean.

The **init** kernel generates *n* datapoints, selecting *k* initial centroids. This results in *n* **assign** kernel instances per age, each fetching a single datapoint and the set of centroids for that age. It then calculates which of these centroids the point is closest to and stores the point in the cluster of that centroid (for the given age). *k* **refine** kernels are then dispatched (per age), each calculating the new mean of that cluster of datapoints, and then stores the result in the Centroids field. This process is then repeated until the algorithm converges.

Evaluation

The *k*-means workload was run on a generated dataset of *n*=2000 with *k*=100. The algorithm was not run until convergence, but stopped after 10 iterations, as the point of convergence is not deterministic.

Microbenchmark

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 zs	9829.00 zs
assign	2024251	4.07 zs	6.95 zs
refine	1000	3.21 zs	92.91 zs
print	11	1.09 zs	379.36 zs

MOTION JPEG

Workload description

Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images.

One **read/splitYUV** kernel instance is dispatched per frame read. It splits the image into its Y, U and V components and stores them in the corresponding fields. Then one kernel per macro block is dispatced, with CIF resolution (352*288) this equals 1584 **yDCT** kernels, 396 **uDCT** and **vDCT** kernels. These kernels then calculate the discreet cosine transform and do quantization, storing their results to their corresponding fields. Then one **VLC/write** kernel per frame is dispatched to perform variable length coding and writing the buffer to disk.

Evaluation

The MJPEG workload was run on the standard test sequence Foreman encoded in CIF resolution. We limited the workload to process 50 frames of video. A single-thread native implementation completed in ~30 sec on the Opteron and ~19 sec on the Core i7.

Microbenchmark

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 zs	18.00 zs
read/splitYUV	51	35.50 zs	1641.57 zs
yDCT	80784	3.07 zs	170.30 zs
uDCT	20196	3.14 zs	170.24 zs
vDCT	20196	3.15 zs	170.58 zs
VLC/write	51	3.09 zs	2160.71 zs

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Paul B. Beskow*, Håvard Espeland*, Håkon K. Stensland*, Preben N. Olsen*,
Ståle Kristoffersen*, Espen A. Kristiansen*, Carsten Griwodz, Pål Halvorsen

*Student author

Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{paulbb, haavares, haakonks, prebenno, staalebk, griff, paalh}@ifi.uio.no

1. INTRODUCTION

As the number of multimedia services grows, so does the computational demands on multimedia data processing. New multi-core hardware architectures provide the required resources, however, parallel, distributed applications are much harder to write than sequential programs. Large processing frameworks like Google's MapReduce [1] and Microsoft's Dryad [2] are steps in the right direction, but they are targeted towards batch processing. As such, we present *P2G*, which is a framework designed to integrate concepts from modern batch processing frameworks into the world of real-time multimedia processing. With P2G we seek to scale transparently with the available resources (following the cloud computing paradigm) and to support heterogenous computing resources, such as GPU processing cores. The idea is to encourage the application developer to express as fine a granularity as possible along two axes, data and functional parallelism, where many of the existing systems sacrifice flexibility in one axis to accommodate for the other, e.g., MapReduce has no flexibility in the functional domain, but allows for fine-grained parallelism in the data domain. In P2G, functional blocks are formulated as kernels that operate on slices of multi-dimensional fields. As such, the fields, used to storing of the multimedia data, are used to express data decomposition. The write-once semantics of the fields provide the needed boundaries and barriers for functional decomposition to exist in our run-time and ensures deterministic output. P2G has intrinsic support for deadlines, and the compiler and run-time analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring. We have implemented a prototype of a P2G execution node, with MJPEG as a primary workload. As such, we intend to demonstrate an operating execution node of P2G at the conference. We will also present results from running tests on our execution node using MJPEG and K-means, where we are able to show that P2G scales with the number of cores available.

2. ARCHITECTURE

As shown in figure 1, P2G consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (i.e., multi-core, GPU, etc) to the master node, which combines this information to form a global topology of available resources. As such, the global topology can change during run-time as execution nodes can be dynamically added and removed to accommodate for changes in the global load.

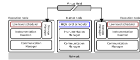


Figure 1: Overview of nodes in the P2G system.

As the master node receives workloads, it use its high-level scheduler to determine which execution nodes to delegate partial or complete parts of the workload to. This process can be achieved in a number of ways. However, as a workload in P2G forms an implict dependency graph based on its store and fetch operations to virtual fields, the high-level scheduler can utilize graph partitioning algorithms, or similar, to map such an implicit dependency graph to the global topology. The utilization of available resources is thus maximized.

When an implicit graph is split across multiple execution nodes, communication is achieved through an event-based, distributed publish-subscribe model. For every input, these subscriptions are deterministically derived from the code and the high-level schedulers partitioning decisions. The subscriptions also make it possible to establish direct communication links between the interacting execution nodes.

P2G uses a low-level scheduler at each execution node to maximize the local scheduling decisions, i.e., the low-level scheduler can decide to combine functional and data decomposition to minimize overhead. During run-time the master node will collect statistics on resource usage from all execution nodes, which all run an instrumentation daemon to acquire this information. The master node can then combine this run-time instrumentation data with the implicit dependency graph derived from the source code and the global topology to make continuous refinements to the high-level scheduling decisions. As such, P2G relies on its combincation of a high-level scheduler, low-level schedulers, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

As seen in figure 2, P2G provides a kernel language for the programmer to write their application in, which they do

(a) Kernel and virtual field definitions

(b) Implicit dependency graph

(c) Kernel and field instances

Figure 2: P2G programming model

by writing isolated, sequential pieces of code called *kernels*. Kernels operate on slices of *fields* through *fetch* and *store* operations and have native code embedded within them. In the model we encourage the programmer to specify the inheret parallelism in their application in as fine a granularity as possible in the domains of functional and data decomposition, without needing to sacrifice the one for the other.

The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to *fetch* slices of data in as fine granularity as possible. The write-once semantics of the fields provide deterministic output, though not necessarily deterministic execution of individual kernels. Given write-once semantics, iteration is supported in P2G by introducing the concept of aging, as seen in figure 2(b), where storing and fetching to the same field position, at different ages, makes it possible to form loops. The write-once semantics also provide natural boundaries and barriers for functional decomposition, as the low-level scheduler can analyze the dependencies of a kernel instance to determine if it is ready for execution. Furthermore, the compiler and the run-time, can analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring.

Given a workload specified using the P2G kernel language, P2G is designed to compile the source code for a number of heterogenous architectures, though it currently only does so for the x86 architecture. P2G can then distribute this workload across the resources available to it.

At the time of writing, P2G consists of what we call an execution node, which is capable of executing entire workloads on a single x86 multi-core node. As such, the high-level scheduler and distribution mechanisms are not yet implemented, though the work is well under way.

3. WORKLOAD

We have implemented a few simple workloads used in multimedia processing to test the prototype implementation, here we will focus on our Motion JPEG implementation.

Figure 3: Overview of the P2G MJPEG encoding process

The *read + splitYUV* kernel reads the YUV-input video and stores the data in three fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing to the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 3, we see that the respective DCT kernels are dependent on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*. Finally, the *VLC + write* kernel does variable length coding and store the MJPEG bit-stream to disk.

4. POSTER & DEMO

In this poster/demo, we will explain and discuss the P2G ideas for multimedia processing with deadlines. Additionally, we accompany the poster with a demo of several well known multimedia workloads and show the entire application development and processing pipeline, i.e, the code in kernel language, the compilation with parallel code generation and the processing automatically distributing the multimedia load to the available processing cores.

5. REFERENCES

[1] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI* (2004), pp. 10–10.

[2] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys* (New York, NY, USA, 2007), ACM, pp. 59–72.

Paper X

P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Håvard Espeland, Paul B. Beskow, Håkon K. Stensland, Preben N. Olsen, Ståle Kristoffersen, Carsten Griwodzand Pål Halvorsen

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2011 International Conference on Parallel Processing Workshops, ed. by Jang-Ping Sheu and Cho-Li Wang, pp. 416-426, IEEE, 2011

Evaluation: 360 papers submitted, 81 accepted, with acceptance rate of 21.0%

Author contribution: Beskow and Espeland were the primary investigators and contributors to this paper.

P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Håvard Espeland, Paul B. Beskow, Håkon K. Stensland, Preben N. Olsen,
Ståle Kristoffersen, Carsten Griwodz, Pål Halvorsen

Department of Informatics, University of Oslo, Norway
Simula Research Laboratory, Norway
Email: {haavares, paulbb, haakonks, prebenno, staalebk, griff, paalh}@ifi.uio.no

***Abstract*—The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multi-core hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counter-part. For this reason, Google and Microsoft implemented their respective processing frameworks MapReduce [10] and Dryad [19], as they allow the developer to think sequentially, yet benefit from parallel and distributed execution. An inherent limitation in the design of these *batch* processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs, or even pre-determined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution.**

With the Nornir runtime system for parallel programs [39], which is a Kahn Process Network implementation, we addressed and solved several of these limitations. However, it is more difficult to use than other frameworks due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, called *P2G*, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches and deadlines, and provides both data- and task-parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We have implemented an (interchangeable) P2G *kernel language* to ease development. In this paper, we present a proof of concept implementation of a P2G execution node and some experimental examples using complex workloads like Motion JPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

I. Introduction

Live, interactive multimedia services are steadily growing in volume. Interactively refined video search, dynamic participation in video conferencing systems and user-controlled views in live media transmissions are a few examples of features that future consumers will expect when they consume multimedia content. New usage patterns, such as extracting features in pictures to identify objects, calculation of 3D depth information from camera arrays, or generating free-view videos from multiple camera sources in real-time, add further magnitudes of processing requirements to already computationally intensive tasks like traditional video encoding. This fact is further exacerbated by the advent of high-definition videos.

Many-core systems, such as graphic processor units (GPUs), digital signal processors (DSPs) and large scale distributed systems in general, provide the required processing power, but taking advantage of the parallel computational capacity of such hardware is much more complex than single-core solutions. In addition, heterogeneous hardware requires individual adaptation of the code, and often involve domain specific knowledge. All this places additional burdens on the application developer. As a consequence, several frameworks have emerged that aim at making distributed application development and processing easier, such as Google's MapReduce [10] and Microsoft's Dryad [19]. These frameworks are limited by their design for batch processing of large amounts of data, with few dependencies across a large cluster of machines. Modifications and enhancements that address bottlenecks [8] together with support for new types of workloads and additional hardware exist [9], [16], [31]. It is also worth mentioning that new languages for current batch frameworks have been proposed [29], [30]. However, the development and processing of distributed multimedia applications is inherently more difficult. Multimedia applications also have stricter requirements for flexibility. Support for iterations is essential, and knowledge of deadlines is often imperative. The traditional batch processing frameworks do not support this.

In our Nornir runtime system for parallel processing [39], we addressed many of the shortcomings of the batch processing frameworks. Nornir is based on the idea of Kahn Process Networks (KPN). Compared to MapReduce-like approaches, Nornir adds support for arbitrary processing graphs, deterministic execution, etc. However, KPNs are designed with some unrealistic assumptions (like unlimited queue sizes), and the Nornir programming model is much more complex than that of frameworks like MapReduce and Dryad. It demands that the application developer establishes communication channels manually to form the dependency graph.

In this paper, we expand on our visions and present our initial ideas of *P2G*. It is a completely new framework for distributed real-time multimedia processing. P2G is designed

to work on continuous flows of data, such as live video streams, while still maintaining the ability to support batch workloads. We discuss the initial ideas and present a proof-of-concept prototype[1] running on x86 multi-core machines. We present experimental results using concrete multimedia examples. Our main conclusion is that the P2G approach is a step in the right direction for development and execution of complex parallel workloads.

II. Related Work

A lot of research has been dedicated to addressing the challenges introduced by parallel and distributed programming. This has led to the development of a number of tools, programming languages and frameworks to ease the development effort.

For example, several solutions have emerged for simplifying distributed processing of large quantities of data. We have already mentioned Google's MapReduce [10] and Microsoft's Dryad [19]. In addition, you have IBM's System S and accompanying programming language SPADE [13]. Yahoo have also implemented a programming language with their PigLatin language [29], other notable mentions for increased language support is Cosmos [26], Scope [6], CIEL [25], SNAPPLE [40] and DryadLINQ [41]. The high-level languages provide easy abstractions for the developers in an environment where mistakes are hard to correct.

Dryad, Cosmos and System S have many properties in common. They all use directed graphs to model computations and execute them on a cluster. System S also supports cycles in graphs, while Dryad supports non-deterministic constructs. However, not much is known about these systems, since no open implementations are freely available. MapReduce on the other hand has become one of the most cited paradigms for expressing parallel computations. While Dryad and System S use a task parallel model, MapReduce uses a data-parallel model based on keys and values. There are several implementations of MapReduce for clusters [1], multi-core [31], the Cell BE architecture [9], and also for GPUs [16]. Map-Reduce-Merge [8] adds a merge step to process data relationships among heterogeneous data sets efficiently, operations not directly supported by the original MapReduce model. In Oivos [35], the same issues are addressed, but in addition, this system provides a more expressive, declarative programming model. Finally, reducing the layering overhead of software running on top of MapReduce is the goal of Cogset [36] where the processing architecture is changed to increase performance.

An inherent limitation in MapReduce, Dryad and Cosmos is their inability to model iterative algorithms. In addition, the rigid MapReduce semantics do not map well to all types of problems [8], which may lead to unnaturally expressed solutions and decreased performance [38]. The limited support for iterative algorithms has been mitigated in HaLoop [5], a fork of Hadoop optimized for batch processing of iterative algorithms where data is kept local for future iterations of the MR steps. However, the programming model of MapReduce is designed for batch processing huge datasets, and not well suited for multimedia algorithms. Finally, Google's patent on MapReduce [11] may prompt commercial actors to look for an alternative framework.

[1] The P2G source code and workload examples are available for download from http://www.p2gproject.org/.

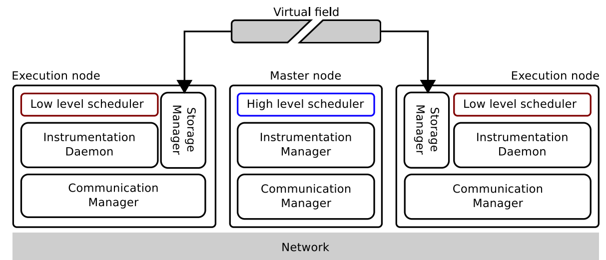


Figure 1. Overview of nodes in the P2G system.

KPN-based frameworks are one such alternative. KPNs support arbitrary communication graphs with cycles and are deterministic. However, in practice, very few general-purpose KPN runtime implementations exist. Known implementations include the Sesame project [34], the process network framework [28], YAPI [22] and our own Nornir [39]. These frameworks have several benefits, but for application developers, the KPN model has some challenges, particularly in a distributed scenario. To mention some issues, a distributed version of a KPN implementation requires a distributed deadlock detection and a developer must specify communication channels between the processes manually.

An alternative framework based on a process network paradigm is StreamIt [15], which comprises a language and a runtime system for simplifying the implementation of stream programs described by a graph that consists of computational blocks (filters) with a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous data-flow process network [23]. The compiler produces code that can make use of multiple machines or CPUs, whose number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

The processing and development of distributed multimedia applications is inherently more difficult than traditional sequential batch applications. Multimedia applications have strict requirements and knowledge of deadlines is necessary, especially in a live scenario. For multimedia applications that enable live communication, iterative processing is essential. Also, elastic scaling with the available resources becomes imperative when the workload, requirements or machine resouces change. Thus, all of the existing frameworks have some short-comings that are difficult to address, and the traditional batch processing frameworks simply come up short in our multimedia scenario. Next, inspired by the strengths of the different approach, we present our ideas for a new framework for distributed real-time multimedia processing.

III. Basic idea

The idea of P2G was born out of the observation that most distributed processing framework lack support for real-time multimedia workloads, and that data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they can severely limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but restrict runtime processing to the domain of data parallelism [12]. Functional languages such as Erlang [3] and Haskell [18] and the event-based SDL [21], map well to task parallelism. Programs are expressed as communicating processes either through message passing or event distribution, which makes it difficult to express data parallelism without specifying a fixed number of communication channels.

In our multimedia scenario, Nornir improves on many of the shortcomings of the traditional batch processing frameworks, like MapReduce and Dryad. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs support also arbitrary communication graphs (with cycles/iterations), while frameworks like MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs). However, Nornir is task-parallel, and data-parallelism must be explicitly added by the programmer. Furthermore, as a distributed, multi-machine processing framework, Nornir still has some challenges. For example, the message-passing communication channels, having exactly one sender and one receiver, are modeled as infinite FIFO queues. In real-life distributed implementations, however, queue length is limited by available memory. A distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually. This requires the developer to think differently than for other distributed frameworks.

With P2G, we build on the knowledge gained from developing Nornir and address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature for processing multimedia workloads includes automatic combined task and data parallelism. Intra-frame prediction in H.264 AVC, for example, introduces many dependencies between sub-blocks of a frame, and together with other overlapping processing stages, these operations have a high potential for benefiting from both types of parallelism. We demonstrated the potential in earlier work with Nornir, whose deterministic nature showed great parallelization potential in processing arbitrary dependency graphs.

Multimedia algorithms being iterative by nature exhibit many pipeline parallel opportunities. Exploiting them are hard because intrinsic knowledge of fine-grained dependences are required, and structuring programs in such a way that pipeline parallelism can be used is difficult. Thies et al. [33] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses assuming that the behaviour is stable. They evaluated their system on multimedia algorithms and gained significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, and this enable the runtime to automatically detect and take full advantage of all parallel opportunities without manual intervention.

A major source of non-determinism in other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. The source of this non-deterministic behavior can be removed by adopting strict write-once semantics for writing to memory [4]. Languages that take advantage of the concept of single assignment include Erlang [3] and Haskell [18]. It enables schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once-semantics are well-suited for a scheduler's dependency analysis, it is not straight-forward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multi-dimensional arrays of data (e.g., pixels in a picture) and provide frequently very intuitive data partitioning opportunities (e.g., 8x8-pixel macro-blocks of a picture). Prominent examples are the computation-heavy MPEG-4 AVC encoding [20] and SIFT [24] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without loosing write-once-semantics.

Flexible partitioning requires the processing of clearly distinct data units without side-effects. The idea adopted for P2G is to use *kernels* as in stream processing [15], [27]. Such a kernel is written once and describes the transformation of multi-dimensional fields of data. Where such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which can analyze dependencies and guarantee fully deterministic output independent of order due to the write-once semantics of fields.

Together, these observations determined four basic ideas for the design of P2G:

- The use of *multi-dimensional fields* as the central concept for storing data in P2G to achieve straight-forward implementations of complex multimedia algorithms.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *write-once semantics* to such fields to achieve deterministic behavior.

- The use of *runtime dependency analysis* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

Within the boundaries of these basic ideas, P2G should be easily accessible for programmers who only need to write isolated, sequential pieces of code embedded in kernel definitions. The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to fetch slices of a field in as fine a granularity as possible, supporting data parallelism.

P2G is designed to be language independent, however, we have defined a C-like language that captures many of P2G's central concepts. As such, the P2G language is inspired by many existing languages. In fact, Cray's Chapel [7] language antedates many of P2G's features in a more complete manner. P2G adds, however, write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements that formulate data dependencies between implicitly instantiated kernels, which are (currently) written in C/C++.

The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations. Furthermore, P2G supports deadlines, which allows scheduling decisions such as termination, branching and the use of alternative code paths based on runtime observations.

In summary, we have opted for an idea that allows programmers to focus on data transformations in a sequential manner, while simultaneously providing enough information for dynamically adapting the data and task parallelization. As an end result of our considerations. P2G's fields look mostly like global multi-dimensional arrays in C, although their representation in memory may deviate, i.e., they need not be placed contiguously in the memory of a single node, and may even be distributed across multiple machines. Although this looks contrary to our message-based KPN approach used in Nornir, it maps well when slices of fields are interpreted as messages and the run-queues of worker threads as KPN channels. An obvious difference is that fields can be read as often as necessary.

IV. Architecture

As shown in figure 1, the P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (a graph of multi-core and single-core CPUs and GPUs, connected by various kinds of buses and other networks) to the master node, which combines this information into a global topology of available resources. As such, the global topology can change during runtime as execution nodes are dynamically added and removed to accommodate for changes in the global load.

To maximize throughput, P2G uses a two-level scheduling approach. On the master node, we have a high-level scheduler (HLS), and on the execution node(s), we use a low-level scheduler (LLS). The HLS can analyze a workloads

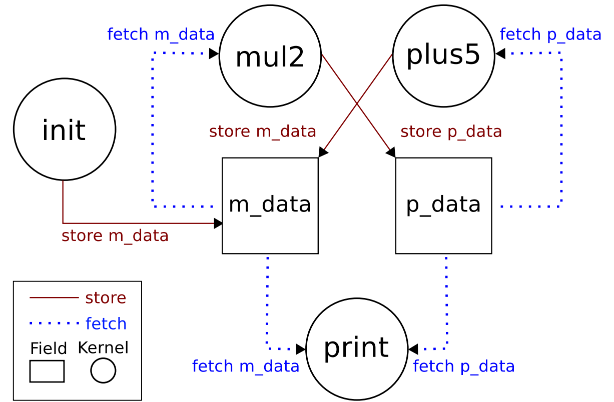


Figure 2. Intermediate implicit static dependency graph

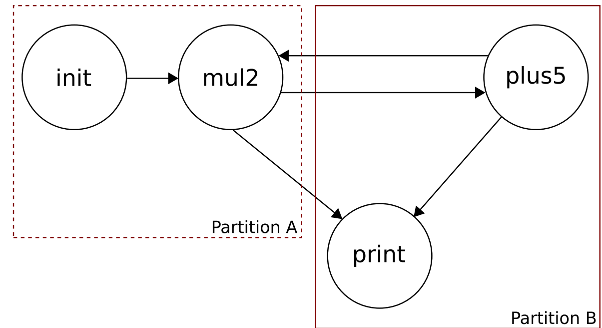


Figure 3. Final implicit static dependency graph

store and fetch statements, from which it can generate an intermediate implicit static dependency graph (see figure 2) where edges connecting two kernels through a field can be merged, circumventing the need for a vertex representing the field (as seen in figure 3). From the intermediate graph, the HLS can then derive a final implicit static dependency graph (see figure 3). The HLS can then use a graph partitioning [17] or search based [14] algorithm to partition the workload into a suitable number of components that can be distributed to, and run, on the resources available in the topology. Using instrumentation data collected from the nodes executing the workload the final graph can be weighted with this profiling data during runtime. The weighted final graph can then be repartitioned, with the intent of improving the throughput in the system, or accommodate for changes in the global load.

Given a partial workload (such as *partition A* from figure 3), an LLS at an execution node is responsible for maximizing local scheduling decisions. We discuss this further in section V, but figure 4 shows how the LLS can combine tasks and data to minimize overhead introduced by P2G, and take advantage of specialized hardware, such as GPUs.

This idea of using a two level scheduling approach is not new. It has also been considered by Roh et al. [32], where they have performed simulations on parallel scheduling decisions for instruction sets of a functional language. Simple workloads are mapped to various simulated architectures, using a "merge-

up" algorithm, which is equivalent to our LLS, and "merge-down" algorithm, which is equivalent to our HLS. These algorithms cluster instructions in such a way that parallelism is not limited, their conclusion is that utilizing a merge-down strategy often is better.

Data distribution, reporting, and other communication patterns is achieved in P2G through an event-based, distributed publish-subscribe model. Dependencies between components in a workload are deterministically derived from the code and the high-level schedulers partitioning decisions, and direct communication occurs.

As such, P2G relies on its combination of a HLS, LLS, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

V. Programming model

The programming model of P2G consists of two central concepts, the *implicit static dependency graph* (figures 2 and 3) and the *dynamically created directed acyclic dependency graph* (DC-DAG) (figure 4). We have also developed a *kernel language* (see figure 5), to make it easier to develop applications using the P2G programming model, though we consider this language to be interchangeable.

The example we use throughout this discussion consists of two primary kernels: *mul2* and *plus5*. These two kernels form a pipeline where *mul2* first multiples a value by 2 and stores this data, which *plus5* then fetches and increases by 5, *mul2* then fetches the data stored by *plus5*, and so on. The *print* kernel runs orthogonally to these two kernels and fetches and writes the data they have produced to *cout*. In combination, these three kernels form a cycle. The kernel *init* runs only once and writes some initial data for *mul2* to consume. The kernels operate on two 1-dimensional, 5 element fields. The print kernel writes *{10, 11, 12, 13, 14}*, *{20, 22, 24, 26, 28}* for the first *age* and *{25, 27, 29, 31, 33}*, *{50, 54, 58, 62, 66}* for the second, etc (as seen in figure 4). As such, the first *iteration* produces the data: *{10, 11, 12, 13, 14}*, *{20, 22, 24, 26, 28}* and *{25, 27, 29, 31, 33}*, and the second *iteration* produces the data: *{50, 54, 58, 62, 66}* and *{55, 59, 63, 67, 71}*, etc. Since there is no termination condition for this program it runs indefinitely.

A. Dependency graphs

The intermediate implicit static dependency graph (as seen in figure 2) is derived from the interaction between fields and kernel definitions, more precisely from the *fetch* and *store* statements of a kernel definition. This intermediate graph can be further refined by merging the edges of kernels linked through a field vertex, resulting in a final implicit static dependency graph, as depicted in figure 3. This final graph can serve as input to the HLS, which can use it to determine how best to partition the workload given a global topology. The graph can be further weighted using instrumentation data, to serve as input for repartitioning. It is important to note that these weighted graphs can serve as input to static offline analysis. For example, it could be used as input to a simulator to best determine how to initially configure a workload, given various global topology configurations.

During runtime, the intermediate implicit static dependency graph is expanded to form a dynamically created directed acyclic dependency graph, as seen in figure 4. This expansion from a cyclic graph to a directed acyclic graph occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iteration. We have chosen to call each such unrolled loop an *Age*. The LLS can then use the DC-DAG to combine tasks and data to reduce overhead introduced by P2G and to take advantage of specialized hardware, such as GPUs. It can then try different combinations of these low-level scheduling decisions to improve the throughput of the system.

We can see how this is accomplished in figure 4. When moving from *Age=1* to *Age=2*, we can see how the LLS has made a decision to reduce data parallelity. In P2G, kernels fetch slices of data, and initially *mul2* was defined to work on each single field entry in parallel, but in *Age=2*, the LLS has decreased the granularity of the fetch statement to encompass the entire field. It could also have split the field in two, leading to two kernel instances of *mul2*, working on disparate sets of the field.

Moving from *Age=2* to *Age=3*, we see how the LLS has made a decision to decrease the task parallelity. This is possible because *mul2* and *plus5* effectively form a pipeline, information that is available from the static graphs. By combining these two tasks, the individual store operations of the tasks are deferred until the data has been fully processed by each task. If the *print* kernel was not present, storing to the intermediate field *m_data* could be circumvented in its entirety.

Finally, moving from *Age=3* to *Age=4*, we can see how a decision to decrease both task and data parallelity has been taken. This renders this single kernel instance effectively instance into a classical *for-loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the data.

P2G makes runtime adjustments dynamically to both data and task parallelism based on the possibly oscillating resource availability and the reported performance monitoring.

B. Kernel language

From our experience with developing Nornir, we came to the realization that expressing workloads in a framework capable of supporting such complex graphs without a high-level language is a difficult task. We have therefore developed a *kernel language*. An implementation of a simple workload is outlined in figure 5, with a C++ equivalent listed in figure 6.

In the current version of our system, P2G is exposed to the developer through this *kernel language*. The language itself is not an integral part and can be replaced easily. However, it exposes several foundations of the P2G design. Most important are the kernel and field definitions, which describe the code and interaction patterns in P2G.

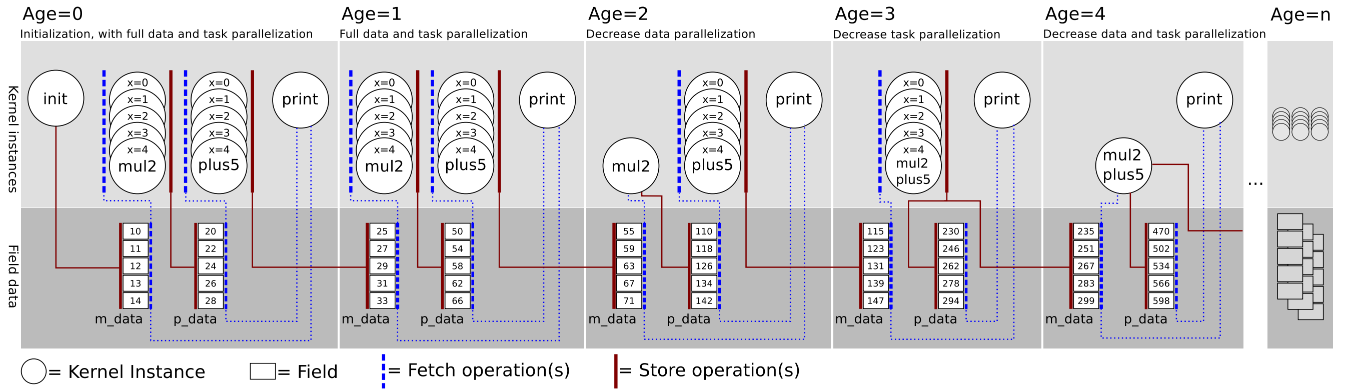


Figure 4. Dynamically created directed acyclic dependency graph (DC-DAG)

Field definitions:

```
0 int32[] m_data age;
1 int32[] p_data age;
```

Kernel definitions:

```
0  init:
1    local int32[] values;
2
3    %{
4      int i = 0;
5      for( ;i < 5; ++i )
6      {
7        put( values, i+10, i );
8      }
9    %}
10
11   store m_data(0) = values;
12
13
```

```
0  mul2:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = m_data(a)[x];
6
7    %{
8      value *= 2;
9    %}
10
11   store p_data(a)[x] = value;
12
13
```

```
0  plus5:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = p_data(a)[x];
6
7    %{
8      value += 5;
9    %}
10
11   store m_data(a+1)[x] = value;
12
13
14
15
```

```
0  print:
1    age a;
2    local int32[] m, p;
3
4    fetch m = m_data(a);
5    fetch p = p_data(a);
6
7    %{
8    for(int i=0; i < extent(m, 0);)
9      cout << get(m, i++) << " ";
10   cout << endl;
11
12   for(int i=0; i < extent(p, 0);)
13     cout << get(p, i++) << " ";
14   cout << endl;
15 %}
```

Figure 5. Kernel and field definitions

A kernel definition's primary purpose is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through the fetch and store statements. As such, a field serves as an interaction point for kernel definitions, as can be seen in figure 2.

An important aspect of multimedia workloads is the ability to express deadlines, where it does not make sense to encode a frame if the playback has moved past that point in the video-stream. Consequently, we have implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer: *timer t1*. This timer can then be polled, and updated, from within a kernel definition, for example *t1+100ms* or *t1 = now*. Given a condition based on a deadline such as *t1+100ms*, a timeout can occur and an alternate code-path can be executed. Such an alternate code-path is executed by storing to a different field then in the primary path, leading to new dependencies and new behavior. Currently, we have basic support for expressing deadlines in the kernel language, but the semantics of these expressions require refinement, as their implications can be considerable.

```cpp
void print( int *data, int num )
{
    for( int i = 0; i < num; ++i )
        std::cout << data[i] << " ";
    std::cout << std::endl;
}
int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    while( true )
    {
        for( int i = 0; i < 5; ++i )
            p_data[i] = m_data[i] * 2;

        print( m_data, 5 );
        print( p_data, 5 );

        for( int i = 0; i < 5; ++i )
            m_data[i] = p_data[i] + 5;
    }
    return 0;
}
```

Figure 6. C++ equivalent of mul/sum example

Fields in P2G have a number of properties, including a type and a dimensionality. Another property is, as mentioned above, *aging*, which allows kernels to be iterative while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times,

as long as the age increases for each store operation (as seen in figure 4). In essence, this adds a dimension to the field and makes it possible to accommodate iterative algorithms. Additionally, it is important to realize that fields are not connected to any single node, and can be fully localized or distributed across multiple execution nodes (as seen in figure 1).

In defining the interaction between kernels and fields, it is encouraged that the programmer expresses the finest possible granularity of kernel definitions, and, likewise, the most precise slices possible for the kernel within the field. This is encouraged because it provides the low-level scheduler more control over the granularity of task and data decomposition. Aided by instrumentation data, it can reduce scheduling overhead by combining several instances of a kernel that process different data, or several instances of different kernels that process data in sequence (as seen in figure 4). The scheduler makes its decisions based on the implicit static dependency graph and instrumentation data.

C. Runtime

Following from the previous discussions, we can extrapolate the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime, and the number of kernel instances executed in parallel for a given kernel definition depends on its fetch statements.

To clarify, a kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements of the kernel definition. For example, looking at figure 4 and 5, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x* fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum possible *x* kernel instances simultaneously per age, giving *a*x mul2* kernel instances. Though, as we have seen, this number can be decreased by the scheduler making *mul2* work over larger slices of data from *m_data*.

With P2G we support implicit resizing of fields, this can be witnessed by looking at the kernel definition of *print* in figure 5. Initially, the extents of *m_data* and *p_data* are not defined, as such, with each iteration of the *for*-loop in *init* the local field *values* is resized locally, leading to a resize of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields impacted by this resize, such as *p_data*. Following the discussion from the previous paragraph, such an implicit resize can lead to additional kernel instances being dispatched.

It is worth noting that a kernel instance is only dispatched when all its dependencies are fulfilled, i.e., that the data it fetches has been stored to the respective fields and elements. Looking at figure 4 and 5 again, we can see that *mul2* stores its result to *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, this means that the kernel instance *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. In our system, each kernel instance is only dispatched once, due to our write-once semantics. To summarize, the *print* kernel instance working on *age=0* becomes runnable when all the elements of *m_data* and *p_data* for *age=0* have been stored. Once it has become runnable, it is dispatched and runs only once.

VI. Prototype Implementation

To verify the feasibility of the P2G framework presented in this paper, we have implemented a prototype version. The prototype consists of a compiler for the kernel language and a runtime that can execute P2G programs on multi-core linux machines.

A. Compiler

Programs written for the P2G system are designed to be platform independent and feature native blocks of code written in C or C++. Heterogeneous systems are specifically targeted, but many of these require a custom compiler for the native blocks, such as nVIDIA's nvcc compiler for the CUDA system and IBM's XL compiler for the Cell Broadband Engine. We decided to compile P2G programs into C++ files, which can be further compiled and linked with native code blocks, instead of generating binaries directly. This approach gives us less control of the resulting object code, but we gain the flexibility and sophisticated optimization of the native compilers, resulting in a lightweight P2G compiler. The P2G compiler works also as a compiler driver for the native compiler and produces complete binaries for programs that run directly on the target system.

B. Runtime

The runtime prototype implements the basic features of a P2G execution node, including multi-dimensional field support, implicit resizing of fields, instrumentation and parallel execution of kernel instances on multiple processors using the implicit dependency graph formed by kernel definitions. However, at the time of writing, the prototype runtime does not yet have a full implementation of deadline expressions, this is because the semantics of the kernel language support for this feature is not fully defined yet.

The prototype targets a node with multiple processors. It is designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which may require resize operations. A kernel subscribes to events related to fields that it depends on, i.e., fields referenced to by the kernels *fetch* statements. When receiving such a storage event, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement, and puts these in a per-kernel ready queue. This means that the ready queues contain always the maximum number of parallel instances that can be executed at any time, only limited by unfulfilled data dependencies.

The low-level scheduler consists of a dependency analyzer and kernel instance dispatcher. Using the implicit dependency graph, the dependency analyzer adds new kernel instances to a ready queue, which later can be processed by the worker

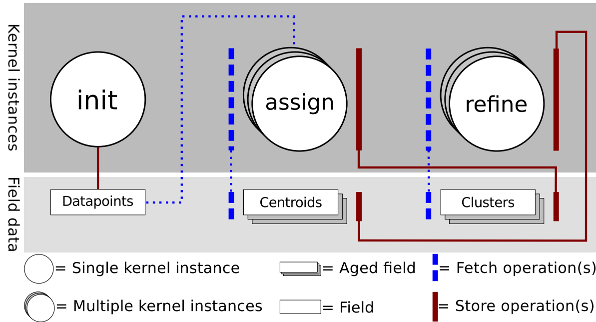


Figure 7. Overview of the K-means clustering algorithm

threads. Dependencies are analyzed in a dedicated thread which handles events emitted from running kernel instances that notifies on *store* and *resize* operations performed on fields. Kernel instances are executed by a worker thread dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements or by groups of kernels that satisfy their own dependencies in aging cycles, such as the *mul2* and *plus5* kernel in figure 5.

The runtime is written in C++ and uses the blitz++ [37] library for high-performance multi-dimensional arrays. The source code for the P2G compiler and runtime can be downloaded from http://www.p2gproject.org/.

VII. Workloads

We have implemented a few workloads commonly used in multimedia processing to test the prototype implementation. The P2G kernel language is able to expose both the data and task parallelism of the programs to the P2G system, so the runtime is able to adapt execution of the programs to suit the target architecture.

A. K-means clustering

K-means clustering is an iterative algorithm for cluster analysis which aims to partition n datapoints into k clusters in which each datapoint belongs to the cluster with the nearest mean. As shown in figure 7, the P2G k-means implementation consists of an *init* kernel, which generates n datapoints and *stores* them to the datapoints field. Then, it selects k of these datapoints randomly, as the initial means, and *stores* them to the centroids field. Next, the *assign* kernel *fetches* a slice of data, a single datapoint per *kernel instance*, the last calculated centroids, and *stores* this datapoint to the cluster of the closest centroids using the euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* form a loop which gradually leads to a convergence in centroids, at which point the k-means algorithm has completed.

B. Motion JPEG

Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images. The MJPEG

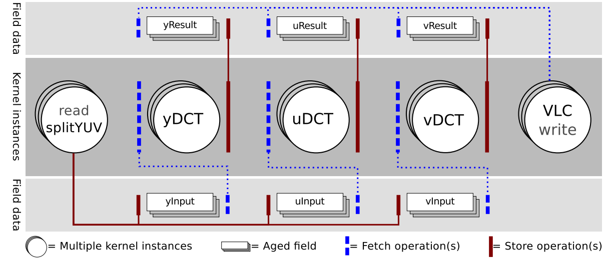


Figure 8. Overview of the MJPEG encoding process

4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table I
Overview of test machines

format provides many layers of parallelism, well suited for illustrating the potential of the framework. We focused on optimizing the discrete cosine transform (DCT) and quantization part as this is the most compute-intensive part of the codec.

The *read + splitYUV* kernel reads the input video in YUV-format and stores the data in three global fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing to the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 8, we see that the respective DCT kernels are dependent on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*. Finally, the *VLC + write* kernel store the MJPEG bit-stream to disk.

VIII. Evaluation

We have run tests with the workloads Motion JPEG and K-means (described in section VII). Each test was run on a *4-way Core i7* and an *8-way Opteron* (see table I for hardware specifications) ranging from *1* worker thread to *8* worker threads with *10* iterations per worker thread count. The results of these tests are reported in the figures 10 and 9, which show the mean running time in *seconds* for each machine for a given thread count with standard deviation reported as error-bars.

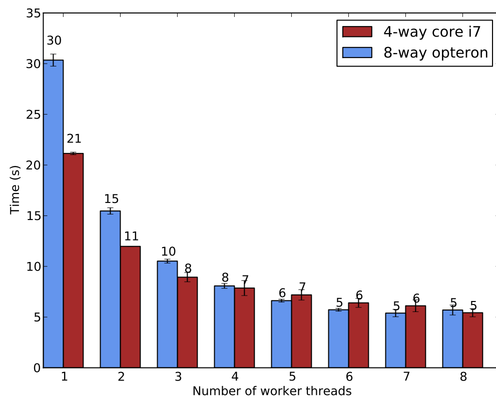


Figure 9. Workload execution time for Motion JPEG

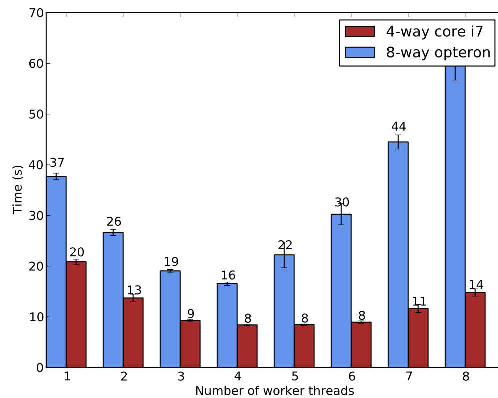


Figure 10. Workload execution time for K-means

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1641.57 μs
yDCT	80784	3.07 μs	170.30 μs
uDCT	20196	3.14 μs	170.24 μs
vDCT	20196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2160.71 μs

Table II
MICRO-BENCHMARK OF MJPEG ENCODING IN P2G

In addition, we have performed micro-benchmarks for each workload, summarized in the tables II and III. The benchmarks summarize the number of kernel instances dispatched per kernel definition, dispatch overhead and time spent in kernel code.

A. Motion JPEG

The Motion JPEG workload is run on the standard test sequence *Foreman* encoded in *CIF* resolution. We limited the workload to process *50* frames of video.

As we can observe from figure 9, P2G is able to scale close to linearly with the resources it has available. In P2G, the dependency analyzer of the LLS runs in a dedicated thread. This affects the running time when moving from *7* to *8* worker threads. Where the eighth thread shares resources with the dependency analyzer. To compare, the standalone single threaded MJPEG encoder on which the P2G version is based upon has a running time of *30* seconds on the Opteron machine and *19* seconds on the Core i7 machine. Note that both the standalone and P2G versions of the MJPEG encoder use a naive DCT calculation, there are versions of DCT that can significantly improve performance, such as FastDCT [2].

From table II, we can see that time spent in kernel code is considerably higher compared to the dispatch overhead for the kernel definitions. The dispatch time includes allocation or reallocation of fields as part of the timing operation. As a result, *init* and *read/splitYUV* have a considerably higher dispatch time then the **DCT* operations.

We can also see that the majority of CPU-time is spent in the kernel instances of *yDCT, uDCT* and *vDCT*, which is the computationally intensive part of the workload. This indicates that decreasing data and task granularity, as discussed in section V-A, has little impact on the throughput of the system. This is because the majority of time is already spent in kernel code.

Note that even though there are *51* instances of the *read-/write* kernel definitions, only *50* frames are encoded, because the last instance reaches the end of the video stream.

B. K-means

The K-means workload is run with K=*100* using a randomly generated data set containing *2000* datapoints. The K-means algorithm is not run until convergence, but with *10* iterations. If we do not define this break-point it is undefined when the algorithm converges, and as such, we have introduced this condition to ensure that we get a relatively stable running time for each run.

As seen in figure 10, the K-means workload scales to *4* worker threads. After this, the running time increases with the number of worker threads. This can be explained by the fine granularity of the *assign* kernel definition, as witnessed when comparing the dispatch time to the time spent in kernel code. This leads to the serial dependency analyzer becoming a bottle-neck in the system. As discussed in section V-A, this condition could be alleviated by decreasing the granularity of data-parallelism, in effect leading to each kernel instance of *assign* working on larger slices of data. By doing so, we would increase the ratio of time spent in kernel code compared to dispatch time and reduce the workload of the dependency analyzer. The reduction in work for the dependency analyzer is a result of the lower number of kernel instances being run.

The two different test machines behave somewhat differently in that the Opteron suffers more than the Core i7 when the dependency analyzer saturates a core. The Core i7 is able to increase the frequency of a single core to mitigate serial bottlenecks, and we think this is why the Core i7 suffers less when we meet the limitations dictated by Amdahl's law.

The considerable time *init* spends in kernel code is because it generates the data set.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9829.00 μs
assign	2024251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table III
MICRO-BENCHMARK OF K-MEANS IN P2G

C. Summary

We have shown that our prototype implementation of an execution node is able to scale with the available resources, as seen in figure 9 and 10. Our initial results indicate that the functionality of decreasing the granularity of task and data parallelity, as discussed in section V-A, is important to ensure full resource utilization.

IX. DISCUSSION

Even though support for deadlines is not yet fully implemented in the P2G runtime, the concept of deadlines formed an integral part of our design goal. The intention behind deadlines is to accommodate for live multimedia workloads, where real-time requirements are mission essential. Varying conditions over time, both in the workload and topology, may effect scheduling decisions: such as termination, branching and the use of alternative code paths based on runtime observations. This is similar to SDL, but unlike contemporary high performance languages.

In P2G, we encourage the programmer to describe the workload in as fine granularity as possible, both in the functional and data decomposition domains. The low-level scheduler has an understanding of both decomposition domains and deadlines. Given this information, the low-level scheduler can minimize overhead by combining functional components and slices of data by adapting to its available resources, be it local cores, or even GPU execution units.

Write-once semantics on fields incurs a large penalty if implemented naively, both in terms of memory usage and data cache misses. However, as the fields are virtual and do not even have to reside in continuous memory, the compiler and runtime are free to optimize field usage. This includes re-using buffers for increased cache locality when old ages are no longer referenced, and garbage collecting old ages. The explicit programming model of P2G allows the system to anticipate what data is needed in the future, which can be used for further optimizations.

Given the complexity of multimedia workloads and the (potentially) heterogeneous resources available in a modern topology, and in many cases, no knowledge of the underlying capabilities of the resources (which is common in modern cloud services), mapping these complex multimedia workloads manually to the available resources becomes an increasingly difficult task, and at some point, even impossible. This is particularly the case where resource availability fluctuates, such as in modern virtual machine parks. With batch processing, where the workloads frequently are not associated with some intrinsic deadline, this task is solved, with frameworks such as MapReduce and Dryad. However, for processing continuous streams such as iterative multimedia algorithms in an elastic manner requires new frameworks; P2G is a step in that direction.

X. CONCLUSION

With P2G, we have proposed a new flexible framework for automatic parallel, real-time processing of multimedia workloads. We encourage the programmer to specify parallelism in as fine a granularity as possible along the axes of data and task decomposition. Using our kernel language this decomposition is expressed through kernel definitions and fetch and store statements on fields. This language is independent from the P2G runtime and can easily be replaced. Given a workload defined in our kernel language it is compiled for execution in P2G. This workload can then be partitioned by the high-level scheduler of a P2G master node, which then distributes partitions to P2G execution nodes which runs the tasks locally. Execution nodes can consist of heterogeneous resources. A low-level scheduler at the execution nodes then adapted the partial (or full) workload to run optimally using resources at hand. Feedback from the instrumentation daemon at the execution node can lead to repartitioning of the workload (a task performed by the high-level scheduler). The aim is to bring the ease of batch-processing frameworks to multimedia workloads.

In this paper we have presented an execution node capable of running on a multi-way architecture. This results from our experiments running on this prototype show the potential of our ideas. However, there still remains a number of vectors for optimization. In the low-level scheduler we have identified that combining task and data to minimize overhead introduced by P2G is a first reasonable modification. Additionally, completing the implementation of a fully distributed version is in the pipeline. Also, writing workloads for heterogeneous processing cores like GPUs and non-cache coherent architectures like Intel's SCC is a further consideration. Currently, we are investigating appropriate mechanisms for both high- and low-level scheduling, garbage collection, fat binaries, resource profiling and monitoring, and efficient migration of tasks.

While a number of optimizations remain, we have determined that P2G is feasible, through the implementation of this execution node, and the successful implementation of multimedia workloads, such as Motion JPEG and k-means. With these workloads we have shown that it is possible to express multimedia workloads in the kernel language and we have implemented a prototype of an execution node in the P2G framework that is able to execute kernels and scales with the available resources.

REFERENCES

[1] Apache. Hadoop, Accessed July 2010. http://hadoop.apache.org.

[2] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of IEICE*, E71(11), 1988.

[3] J. Armstrong. A history of Erlang. In *Proc. of ACM HOTL III*, pages 6:1–6:26, 2007.

[4] R. Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, 1989.

[5] Y. Blu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

[6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[8] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[9] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI*, pages 10–10, 2004.

[11] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. *US Patent Application*, (US 7650331), 2010.

[12] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[14] F. Glover. Tabu search, Part I1. *ORSA journal on Computing*, 2(1):4–32, 1990.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of PACT*, pages 260–269, New York, NY, USA, 2008. ACM.

[17] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing* 1. *Parallel Computing*, 26(12):1519–1534, 2000.

[18] P. H. J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of ACM HOTL III*, pages 12:1–12:55, 2007.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59–72, New York, NY, USA, 2007. ACM.

[20] ISO/IEC. *ISO/IEC 14496-10:2003*, 2003. Information technology - Coding of audio-visual objects - Part 10: Advanced Video Coding.

[21] ITU. *Z.100*, 2007. Specification and Description Language (SDL).

[22] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf. Yapi: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC'2000*, pages 402–405. ACM Press, 2000.

[23] T. Lee, E.A.; Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[24] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[25] D. Murray, M. Schwarzkopf, and C. Smowton. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[26] C. Nicolaou. An architecture for real-time multimedia communication systems. *Selected Areas in Communications, IEEE Journal on*, 8(3):391–400, 1990.

[27] Nvidia. Nvidia cuda programming guide 3.2, Aug. 2010.

[28] A. G. Olson and B. L. Evans. Deadlock detection for distributed process networks. In *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP*, pages 73–76, 2006.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[30] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of IEEE HPCA*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[32] L. Roh, W. A. Najjar, and A. P. W. Böhm. Generation and quantitative evaluation of dataflow clusters. In *ACM FPCA: Functional Programming Languages and Computer Architecture*, New York, NY, USA, 1993. ACM.

[33] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[34] M. Thompson and A. Pimentel. Towards multi-application workload modeling in sesame for system-level design space exploration. In S. Vassiliadis, M. Berekovic, and T. Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222–232. Springer Berlin / Heidelberg, 2007.

[35] S. V. Valvåg and D. Johansen. Oivos: Simple and efficient distributed data processing. In *Proc. of IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 113–122, 2008.

[36] S. V. Valvåg and D. Johansen. Cogset: A unified engine for reliable storage and parallel processing. In *Proc. of IFIP International Conference on Network and Parallel Computing Workshops (NPC)*, pages 174–181, 2009.

[37] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '98, pages 223–230, London, UK, 1998. Springer-Verlag.

[38] Ž. Vrba, P. Halvorsen, C. Griwodz, and P. Beskow. Kahn process networks are a flexible alternative to mapreduce. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:154–162, 2009.

[39] Ž. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen. The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce. *Journal of Sumpercomputing*, 27(1), 2010.

[40] D. Waddington, C. Tian, and K. Sivaramakrishnan. Scalable lightweight task management for mimd processors, 2011.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. Dryadlinq: A system for general-puropose distributed data-parallel computing using a high-level language, 2008.

Paper XI

Processing of Multimedia Data using the P2G Framework

Paul B. Beskow, Håkon K. Stensland, Håvard Espeland, Preben N. Olsen, Ståle Kristoffersen, Carsten Griwodzand Pål Halvorsen

Published: Proceedings of the 19th ACM international conference on Multimedia, ed. by Hari Sundaram, Wu-Chi Feng, Nicu Sebe, pp. 819-820, ACM, 2011

Author contribution: Beskow and Espeland were the primary contributors and investigators of this demonstration implementing the demonstration application and presenting the results.

Processing of Multimedia Data using the P2G Framework

Paul B. Beskow, Håkon K. Stensland, Håvard Espeland, Espen A. Kristiansen,
Preben N. Olsen, Ståle Kristoffersen, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{paulbb, haakonks, haavares, prebenno, espeak, staalebk, griff, paalh}@ifi.uio.no

ABSTRACT

In this demo, we present the *P2G* framework designed for processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches and deadlines. P2G is implemented to scale transparently with available resources, i.e., a concept familiar from the cloud computing paradigm. Additionally, P2G supports heterogeneous computing resources, such as x86 and GPU processing cores. We have implemented an interchangeable P2G *kernel language* which is meant to expose fundamental concepts of the P2G programming model and ease the application development. Here, we demonstrate the P2G execution node using a MJPEG encoder as an example workload when dynamically adding and removing processing cores.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUE**]: Concurrent Programming—*Parallel programming*

General Terms

Languages, Performance

1. INTRODUCTION

As the number of multimedia services grows, so do the computational demand of multimedia data processing. New multi-core hardware architectures provide the required resources, but parallel, distributed applications are much harder to write than sequential programs. Large processing frameworks like Google's MapReduce [1] and Microsoft's Dryad [3] are steps in the right direction, but they are targeted towards batch processing. As such, we present *P2G* [2], a framework designed to integrate concepts from modern batch processing frameworks into the world of real-time multimedia processing. We seek to scale transparently with the available resources (following the cloud computing paradigm) and to support heterogeneous computing resources, such as GPU processing cores. The idea is to encourage the application developer to express the application as fine granular as possible along two axes, i.e., both data and functional parallelism. In this respect, many of the existing systems sacrifice flexibility in one axis to accommodate for the other, e.g., MapReduce has no flexibility in the functional domain, but allows for fine-grained parallelism in the data domain. In P2G, functional blocks called kernels operate on slices of multi-dimensional fields. We use fields to store multimedia data and implicitly express data decomposition. A write-once semantics of the fields provides the needed boundaries and barriers for functional decomposition to exist in our run-time and ensures deterministic output. P2G has intrinsic support for deadlines. Moreover, the compiler and run-time analyze dependencies dynamically and then merge or split kernels based on resource availability and performance monitoring.

We have implemented a prototype of a P2G execution node. Here, we demonstrate an operating execution node of P2G using MJPEG as an example. We are able to show that P2G scales dynamically with the number of available x86 processing cores.

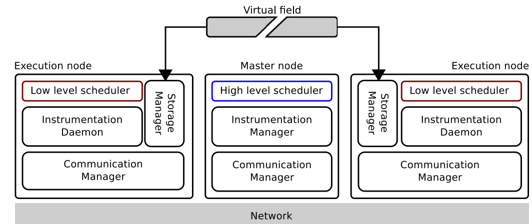


Figure 1: Overview of nodes in the P2G system.

2. THE P2G ARCHITECTURE

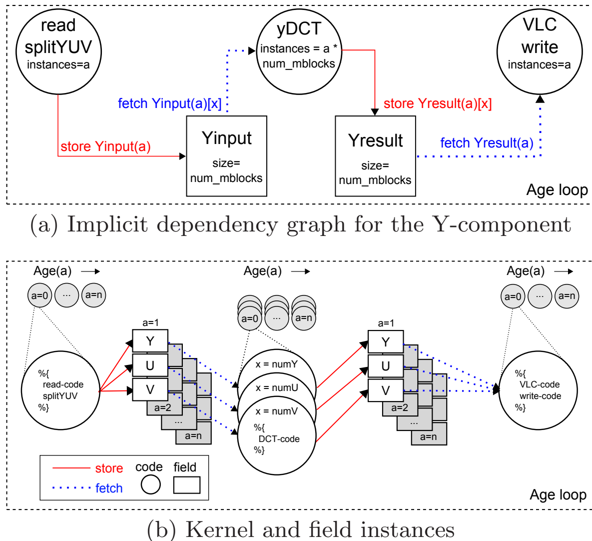
As shown in figure 1, P2G consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (i.e., multi-core, GPU, etc) to the master node, which combines this information to form a global topology of available resources. As such, the global topology can change during run-time as execution nodes can be dynamically added and removed to accommodate for changes in the global load.

As the master node receives workloads, it uses its high-level scheduler to determine which execution nodes to delegate partial or complete parts of the workload to. This process can be achieved in a number of ways. However, as a workload in P2G forms an implicit dependency graph based on its *store* and *fetch* operations, the high-level scheduler can utilize graph partitioning algorithms, or similar, to map such an implicit dependency graph to the global topology. The utilization of available resources can thus be maximized.

P2G uses a low-level scheduler at each execution node to maximize the local scheduling decisions, i.e., the low-level scheduler can decide to combine functional and data decomposition to minimize overhead. During run-time, the

Copyright is held by the author/owner(s).
MM'11, November 28–December 1, 2011, Scottsdale, Arizona, USA.
ACM 978-1-4503-0616-4/11/11.

master node will collect statistics on resource usage from all execution nodes, which all run an instrumentation daemon to acquire this information. The master node can then combine this run-time instrumentation data with the implicit dependency graph derived from the source code and the global topology to make continuous refinements to the high-level scheduling decisions. As such, P2G relies on its combination of a high-level scheduler, low-level schedulers, instrumentation data and the global topology to make best use of the performance of several (possibly heterogeneous) cores in a distributed system.



(a) Implicit dependency graph for the Y-component

(b) Kernel and field instances

Figure 2: Programming model (MJPEG example)

P2G provides a kernel language for the programmer to write their application in, which they do by writing isolated, sequential pieces of code (kernels). Kernels operate on slices of *fields* through *fetch* and *store* operations and have native code embedded within them. In this model, we encourage the programmer to specify the inherent parallelism in their application in as fine a granularity as possible in the domains of both functional and data decomposition.

The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to *fetch* fine granular data slices. A write-once semantics of the fields provides deterministic output, though not necessarily deterministic execution of individual kernels. Given write-once semantics, iteration is supported in P2G by introducing the concept of aging, as seen in figure 2(a), where storing and fetching to the same field position, at different ages, makes it possible to form loops. The write-once semantics also provides natural boundaries and barriers for functional decomposition, as the low-level scheduler can analyze the dependencies of a kernel instance to determine if it is ready for execution. Furthermore, the compiler and the run-time can analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring.

Given a workload specified using the P2G kernel language, P2G is designed to compile the source code for a number of heterogeneous architectures. At the time of writing, P2G consists of what we call an execution node, which is capable of executing workloads on a single x86 multi-core node. As such, the implementation of a high-level scheduler, support

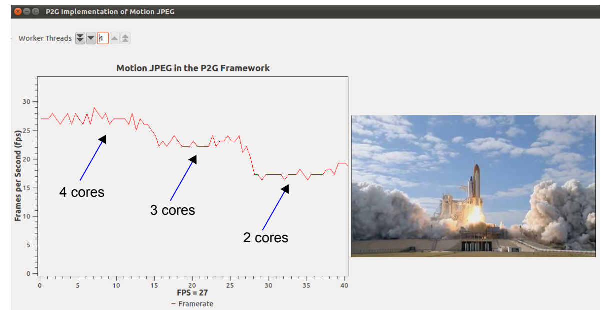


Figure 3: Screenshot of the MJPEG demo.

for heterogeneous architectures and distribution mechanisms are ongoing work.

3. WORKLOAD

We have implemented several simple workloads used in multimedia processing, such as matrix multiplication, k-means clustering to test the prototype implementation of P2G. In this demo, we will focus on our implementation of the MJPEG encoder.

Figure 2(b) shows how the MJPEG encoding process is split into kernels and their running instances. YUV-input video are read into 8x8 macro-blocks and stored in three global fields, by the *read + splitYUV* kernel. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the *DCT* kernel transforming the luminance component of the picture. Each of these kernel instances stores the DCT'ed macro-block into global result fields. Finally, the *VLC + write* kernel does variable length coding and stores the MJPEG bit-stream to disk.

4. DEMONSTRATION

In this demo, we will explain and discuss the P2G ideas for multimedia processing. A screenshot of the MJPEG encoder running in the P2G framework is shown in figure 3. We are able to dynamically add and remove processing cores available to the framework. To show the performance, we live plot the achieved frame-rate from the encoder (left) and show a preview of the encoded video stream (right) as seen in the screenshot.

Acknowledgements

This work has been performed in the context of the *iAD* centre for Research-based Innovation (project number 174867) funded by the Norwegian Research Council.

5. REFERENCES

[1] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI* (2004), pp. 10–10.

[2] Espeland, H., Beskow, P. B., Stensland, H. K., Olsen, P. N., Kristoffersen, S., Griwodz, C., and Halvorsen, P. P2G: A framework for distributed real-time processing of multimedia data. In *Proc. of SRMPDS 2011* (2011), IEEE.

[3] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys* (2007), ACM, pp. 59–72.