Doctoral Dissertation

# Implementation and performance aspects of Kahn process networks

## An investigation of problem modeling, implementation techniques, and scheduling strategies

Željko Vrba

July 2009

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo

Non quia difficilia sunt non audemus, sed quia non audemus,
difficilia sunt.

## Abstract

The appearance of commodity multi-core processors, has spawned a wide interest in parallel programming, which is widely-regarded as more challenging than sequential programming. KPNs are a model of concurrency that relies exclusively on message passing, and that has some advantages over parallel programming tools in wide use today: simplicity, graphical representation, and determinism. Because of determinism, it is possible to reliably reproduce faults, an otherwise notoriously difficult problem with parallel programs. KPNs have gained acceptance in simulation and signal-processing communities. In this thesis, we investigate the applicability of KPNs to implementing general-purpose parallel computations for multi-core machines. In particular, we investigate 1) how KPNs can be used for modeling general-purpose problems; 2) how an efficient KPN run-time can be implemented; 3) what KPN scheduling strategies give good run-time performance.

For these purposes, we have developed Nornir, an efficient run-time system for executing KPNs. With Nornir, we show that it is possible to develop a high-performance KPN run-time for multi-core machines. We experimentally demonstrate that problems expressed in the Kahn model resemble very much their sequential implementations, yet perform much better than when expressed in the MapReduce model, which has become widely-recognized as a simple parallel programming model. Lastly, we use Nornir to evaluate several load-balancing methods: static assignment, work-stealing, our improvement of work-stealing, and a method based on graph partitioning. The understanding brought by this evaluation is significant not only in the context of the Kahn model, but also in the more general context of load-balancing (potentially distributed) applications written in message-passing style.

# Preface

> "I do not ask you about fifteen days ago. But what about fifteen days hence? Come, say a word about this!" Since none of the monks answered, he answered for them: "Every day is a good day."
>
> *(Ummon)*

This thesis is submitted to the Department of Informatics, Faculty of Mathematical and Natural Sciences, University of Oslo, Norway, as a partial requirement for the degree philosophiæ doctor. The work has been carried out in the period from march 2005 to july 2009 with financial support from the Research Council of Norway and Department of Informatics.


## Acknowledgments

PhD. . . a unique, indescribable experience, shaped to a large degree by the people one meets. I thank. . .

. . . Erek Göktürk, for his technical insights and for countless thought-provoking conversations of the non-technical nature.

. . . Asbjørn Sannes, for making a long period of loneliness bearable by being (good!) company during lunch-breaks.

. . . employees at IFI's administration, for always being helpful and making things easier. I thank in particular Lena Korsnes, my PhD study adviser, and Line Valbø, who always found some time for listening to whatever was bothering me at the occasion.

. . . Aslak Johannesen, whom I wish I had gotten to know better, for asking the right question in the moment of doubt.

. . . master- and PhD-students at Networks and distributed systems group of Simula Research Laboratory, for just being there and making the "office" a cheerful workplace.

. . . last, but not least, my supervisors Pål Halvorsen and Carsten Griwodz, for believing in me, supporting me, and pushing me over the edges.

# Contents

**1. Introduction** **1**
1.1. Motivation . . . 1
1.2. Problem statement . . . 3
1.3. Contributions . . . 3
1.4. Synopsis . . . 5

**2. Background material** **7**
2.1. Parallel programming tools . . . 7
2.1.1. Dichotomies . . . 7
2.1.2. Tool support for task parallelism . . . 10
2.1.3. Large-scale frameworks . . . 16
2.2. Operating system facilities . . . 16
2.2.1. Processes and threads . . . 17
2.2.2. Scheduling . . . 17
2.2.3. Inter-process communication . . . 19
2.2.4. Process and thread synchronization . . . 21
2.2.5. Deadlock detection . . . 23
2.3. Summary . . . 23

**3. Modeling with Kahn process networks** **25**
3.1. Semantics of Kahn process networks . . . 25
3.1.1. Operational semantics . . . 25
3.1.2. Denotational semantics . . . 27
3.1.3. KPNs and coroutines . . . 27
3.1.4. Limitations of determinism . . . 28
3.1.5. A practical problem: execution in limited space . . . 30
3.2. Relation to known patterns . . . 33
3.2.1. Pipelines . . . 34
3.2.2. Parallel pipelines . . . 34
3.2.3. Distributed parallel pipeline . . . 35
3.3. Case studies . . . 36
3.3.1. An analysis of MapReduce . . . 36
3.3.2. Word frequency . . . 39
3.3.3. K-means . . . 41

3.4. Executing KPNs by iterative MapReduce . . . 45
3.4.1. Pure solution . . . 48
3.4.2. Impure solution . . . 50
3.5. Existing KPN implementations . . . 51
3.6. Summary . . . 52

**4. Nornir implementation 55**
4.1. Process management and scheduling . . . 55
4.1.1. Data structures . . . 56
4.1.2. User-mode context switch . . . 58
4.1.3. Control flow . . . 60
4.2. Load-balancing methods . . . 63
4.2.1. Static assignment . . . 63
4.2.2. Work stealing . . . 64
4.2.3. Graph partitioning . . . 68
4.3. Message passing . . . 72
4.4. Deadlock detection . . . 73
4.4.1. Blocking graph . . . 74
4.4.2. Integration with Nornir . . . 74
4.5. Configurability and portability . . . 75
4.5.1. Scheduling options . . . 75
4.5.2. Accounting options . . . 75
4.5.3. Run-time options . . . 77
4.5.4. Abandoned options . . . 77
4.6. Summary . . . 78

**5. Performance evaluations 81**
5.1. Application benchmarks . . . 81
5.1.1. Methodology . . . 81
5.1.2. Word frequency . . . 82
5.1.3. k-means . . . 84
5.1.4. Comparison with Phoenix . . . 85
5.1.5. Conclusions . . . 87
5.2. Microbenchmarks . . . 88
5.2.1. Workloads . . . 89
5.2.2. Performance of implementation options . . . 94
5.2.3. Performance evaluation of Nornir . . . 101
5.2.4. Conclusions . . . 112
5.3. Evaluation of load-balancing methods . . . 114
5.3.1. Comparison of work-stealing and graph-partitioning . . . 114
5.3.2. Limits of work stealing . . . 124
5.3.3. Conclusions . . . 135
5.4. Summary . . . 136

**6. Conclusion and further directions** 139
6.1. Review of problem statement and contributions . . . . . . . . . . . . . . 140
6.2. A critical review . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
6.3. Future directions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143

**Bibliography** 145

**A. Review of graph theory concepts** 155

**B. Graph cuts and partitions** 157
B.1. The k-cut problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
B.1.1. Maximum flow and (2-)cut of a graph . . . . . . . . . . . . . . . . . 157
B.1.2. Saran-Vazirani algorithm . . . . . . . . . . . . . . . . . . . . . . . . 158
B.2. Graph bisection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
B.2.1. KL heuristics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
B.2.2. FM heuristics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
B.3. The k-partition problem . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
B.3.1. Theoretical bounds . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
B.3.2. Recursive bisection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
B.3.3. Stochastic methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
B.3.4. Multilevel and greedy heuristics . . . . . . . . . . . . . . . . . . . . 161
B.4. Mapping problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162

**C. Computer architecture** 163
C.1. Memory subsystem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
C.2. Performance effects of memory organization . . . . . . . . . . . . . . . . 164

**D. A complete code example** 167

# 1. Introduction

We are witnessing a small technological revolution. As modern processors have reached their power and frequency limits, the designers are turning to increasing the on-chip parallelism. The trend began with the introduction of IBM's dual-core POWER4 processor [1, 2], followed by Intel's hyperthreading. Intel and AMD have introduced dual- and quad-core CPUs, making them today (2009) a commodity; on the high-end there is Sun's Niagara T2 chip [3] which is capable of supporting 8 hardware threads on each of the 8 cores (64 threads in total). In order to reduce power dissipation, individual cores are running on lower frequencies than that of their single-core predecessors. Consequently, the *sequential* performance of individual cores is decreased, so developers cannot any longer develop sequential applications and count on that they will execute faster on future-generation CPUs. High-performance applications will therefore have to be designed with parallelism in mind from the beginning. We thus need abstractions that will ease development and modeling of parallel applications, as well run-time environments that can efficiently support these abstractions.

## 1.1. Motivation

A historical study of programming languages reveals many attempts to define suitable abstractions for parallel programming, such as futures [4] and monitors [5]. Despite research efforts, lower-level programming languages, such as C and C++, have no support for concurrency in the core language. Instead, they support concurrency only through system libraries such as the pthread interface standardized by POSIX. These libraries provide low-level synchronization primitives, such as mutexes, condition variables and semaphores, so developers must themselves build higher-level abstractions. Furthermore, these primitives are hard to use correctly even for experienced programmers [6]; their incorrect use leads to problems such as deadlocks, race conditions, or priority inversions. Newer languages, however, *do* recognize concurrency as a first-class concept: for example, Java uses the `synchronized` keyword to provide partial [7] support for monitors, and C# uses the `lock` keyword for mutual exclusion.

A trait common to the above-mentioned mechanisms is that they are *non-deterministic*, which may lead to the (most often) undesireable situation where the application's results may *differ* between runs with the *same* data. Because of non-determinism, these mechanisms are hard and unintuitive to use – for example, many developers with little experience in parallel programming expect that mutexes unblock waiting threads in FIFO order, which is not so in most implementations. Non-determinism is also the main cause that makes reproducing and correcting concurrency bugs so notoriously

hard. Lee [6] therefore argues that, contrary to the today's situation, deterministic behavior should be *default*, and that non-deterministic behavior should be possible when explicitly desired.

Industrial actors have met their needs for simplified concurrent and distributed programming by developing their own solutions. Google has developed MapReduce [8], Yahoo has developed Pig latin programming language [9], while Microsoft has developed Dryad [10], Cosmos and programming language Scope [11]. These systems are developed with distributed, large-scale data analyses and transformations in mind. Their common traits are the use of *shared-nothing* programming model, i.e., explicit message-passing for communication between concurrent components, and that deterministic behavior is default. These properties benefit program development in several ways:

- *Sequential coding of individual components.* Processes are written in the usual sequential manner where synchronization is implicit in communication primitives (message send and receive). Developers can thus reuse their existing expertise, and need not worry about non-deterministic aspects of concurrent execution.
- *Preserved composability.* Determinism guarantees that connecting the outputs of components computing functions $f(x)$ and $g(x)$ to the inputs of a component computing function $h(x, y)$ will result in $h(f(x), g(x))$. Thus, components can be developed and tested *individually*, and later assembled into more complex programs.
- *Transparent parallelization and distribution.* The actual mechanisms for achieving parallelization are hidden from developers. Components are written from scratch in the message-passing model, so they can run *unmodified* on multi-core machines or on large distributed clusters, perhaps under a different run-time.
- *Reliable reproduction of program faults*, an otherwise notoriously difficult problem with concurrent systems, is possible when developers restrict themselves to the deterministic subset of the framework. Of the above frameworks, Dryad supports also non-deterministic constructs.

All of the above systems lack support for a feature that is actually very useful: feedback loops in the application's data path. In other words, these frameworks are designed with *one-pass* data transformations in mind. On the other hand, there are many problems where feedback loops in the data-path are essential. Examples of such programs are video-encoding and iterative algorithms where the number of iterations cannot be determined *a priori*, e.g., the k-means classification algorithm or N-body simulations.

The Kahn process network (KPN) model [12] is an asynchronous, deterministic, and shared-nothing concurrent programming model that *does* support feedback loops. Thus, along with the above properties, the Kahn model also provides additional *flexibility*. The combination of asynchronous communication and support for feedback loops allow developers to express certain patterns, such as mutual recursion, which are not expressible in systems without these properties.

We know of only two other systems with support for feedback loops: IBM's System S and its programming language SPADE [13], which have been developed in parallel with our work. However, information about its architecture and implementation is scarce, and the system is not easily available. The other system is StreamIt [14], which is more restricted than KPNs, as we shall explain in section 2.1.2.2.

## 1.2. Problem statement

Because of the properties mentioned above, KPNs are already used in embedded, signal-processing and simulation domains [15, 16, 17, 18, 19, 20]. In this thesis, we experimentally investigate their applicability to parallelizing general-purpose applications on multi-core, shared-memory machines. In particular, we address the following questions:

1. How can we make an efficient run-time system for executing KPNs on multi-core machines? (Distributed implementation is left for future work.)
2. Are KPNs applicable to parallelizing general-purpose applications and what is the required effort of doing so?
3. What are the performance characteristics of applications implemented within the KPN framework?
4. What is the effect of KPN scheduling policies on application performance?

Experimentation presupposes the existence of a run-time system for executing KPNs. For answering the above questions, such system must fulfill several requirements. First, it must support *multiprocessing* and run-time *deadlock detection and resolution*, which is a "must-have" feature that enables execution of arbitrary KPNs (within the limits of available memory). Second, it must support *easy experimentation* with low-level mechanisms, such as scheduling and message-passing, so that their relative merits can be easily judged. Third, it must support collection of *detailed accounting data*. Accounting is necessary for quantifying overheads, as well as for implementing scheduling policies that take into account more factors than just CPU consumption of individual KPs. Fourth, it should efficiently support *overdecomposed applications*, i.e., those having more many more Kahn processes (KPs) than CPUs. Efficient support for overdecomposition enables fine-grained parallelism, which leads to transparent speedup on machines with more CPUs.

## 1.3. Contributions

Since none of the existing KPN run-time implementations [20, 19, 15, 21, 22] satisfy all of our requirements, we have implemented Nornir [23, 24, 25],[1] which is a configurable,

[1]Nornir (pl.) spin the threads of fate at the foot of Yggdrasil, the tree of the world. [Source: Wikipedia]

multi-platform and efficient run-time environment for executing KPNs on multi-core machines. We have estimated that a clean-start implementation would not take considerably longer time than implementing the missing functionality for an existing run-time. A clean-start implementation also removes the risks of encountering bugs in a large and unfamiliar code-base.

To answer the first question, we have used Nornir to evaluate performance characteristics of message-passing and scheduling options. We have implemented an optimized message-passing mechanism and found that its overheads are smaller by a factor of 1.7–1.9 than the overheads of POSIX message queues. We have also implemented a user-space m:n KPN scheduler and found that its overheads are smaller by a factor of up to 6.5 than the overheads of the kernel's thread scheduler. We have measured that the cost of a single message send/receive operation combined with a context switch takes $\sim 0.8\mu$s. Chapter 4 and section 5.2 extend the results published in [23].

To answer the second and third questions, we have first thoroughly analyzed the MapReduce semantics and found that it can be easily implemented within the KPN framework. We have then used Nornir to implement two real-world problems, word frequency and k-means, as KPNs in two ways: one that "naturally" models the problem and another that faithfully implements the MapReduce semantics. Our experiments show that the "natural" KPN outperforms the "MapReduce" KPN by a factor of 2.95 – 6.74 on the word frequency application, and by a factor of 1.17 – 1.43 on the k-means application. Furthermore, we have evaluated the "natural" solutions against their counterparts in Phoenix [26], which is a MapReduce implementation optimized for multi-core architectures. Experiments show that our "natural" solutions in Nornir outperform the Phoenix solutions by factors of 2.65 – 2.76 and 1.28 – 1.75, respectively. Sections 3.2, 3.3, and 5.1 extend the discussions and results published in [24].

Our case-studies are focused on MapReduce for several reasons. First, MapReduce is a widely recognized "simple" framework for parallel programming, and many developers are already familiar with it. Second, many have reimplemented MapReduce for different architectures [27, 26, 28, 29] and made it publicly available. The availability Phoenix made it possible to put our results in perspective with similar research. We have not focused on Dryad, Cosmos or System S because their models are already very general and similar to KPNs – indeed, deterministic programs built for these systems[2] *are* KPNs. Furthermore, very little information is available about these systems, and they are not easily available to the general public, if at all.

To answer the fourth question, we have designed synthetic benchmarks and evaluated their performance under the established work-stealing [30] scheduling algorithm, and the algorithm proposed by Devine et al. [31, 32]. Devine's algorithm is designed for load-balancing of structured distributed applications, such as simulations. It attempts to achieve load-balance across machines, while simultaneously reducing communication volume between processes running on different machines and costs of process migration. We believe to be the first who have evaluated performance of load-balancing algorithms based on graph partitioning, such as Devine's, on unstructured workloads and

[2]Recall that Dryad supports also non-deterministic constructs.

multi-core machines. Our results show that work-stealing has superior performance,[3] and they uncover a serious problem with Devine's algorithm on our workloads: large variance of program running times, an aspect not discussed by Devine. We have also investigated the scalability of the work-stealing algorithm along two axes: number of KPs in the network and granularity of work division among them. Sections 4.2 and 5.3 extend the discussions and results published in [25].

During our research, we have also investigated two additional topics, not originally addressed by our questions. First, we have devised two constructions, described in section 3.4, that enable execution of KPNs by repeatedly iterating MapReduce computations. The benefit of these constructions is that they provide a way of executing KPNs in a distributed and fault-tolerant manner until such KPN run-time is developed. Furthermore, they are immediately implementable, as open-source MapReduce implementations are readily available [27]. Second, we have found a simple modification to the work-stealing algorithm, described in section 4.2.2.4, that significantly improves performance of some workloads. Performance comparison with the original work-stealing algorithm is given in section 5.3.2.

## 1.4. Synopsis

This thesis is organized as follows. Chapter 2 gives background information. We elaborate on forms of concurrency, formal models of parallel computation that are similar to KPNs, and support for concurrency in programming languages. We also discuss operating-system facilities related to KPN implementation: scheduling, communication and synchronization mechanisms.

Chapter 3 is dedicated to modeling applications using KPNs. There, we describe the KPN semantics, discuss existing KPN implementations and draw parallels with existing patterns. In particular, we focus on MapReduce [8] and show three different ways of casting its semantics into the KPN framework. We then use the typical MapReduce examples, word frequency and k-means, to show that they can be more naturally expressed in the KPN framework. Finally, we present two ways of executing general KPNs by using *iterative* MapReduce computations, which might be useful where distribution and fault-tolerance are needed.

In chapter 4 we describe Nornir implementation in depth, focusing on our implementation of m:n scheduling, load-balancing, message-passing and deadlock detection mechanisms. We have implemented three load-balancing algorithms, static assignment, work-stealing [30] and a method based on graph-partitioning devised by Devine et al. [31, 32], which we have adapted to scheduling KPNs. The last method attempts to achieve load-balance across CPUs, while simultaneously reducing communication volume between KPs running on different CPUs and costs of KP migration. In chapter 4 we also describe Nornir's accounting mechanisms, configuration options and discuss

[3]For simplicity of expression, we shall often talk about relative performance of scheduling algorithms. By this, we will actually mean performance of *applications* running under the scheduling algorithms in question.

ideas that we have investigated and abandoned.

Chapter 5 is dedicated to performance evaluations. In the first part of the chapter, we visit our case-study applications, word frequency and k-means, and show that they indeed do perform better when modeled with KPNs than when modeled with MapReduce. The second part of the chapter is dedicated to microbenchmarks. We first describe our benchmark workloads, and then use them to investigate performance of the KPN implementation options as well as the performance and scalability of Nornir in the optimal configuration. In the third part of the chapter, we evaluate performance of work-stealing and Devine's load-balancing algorithms.

Chapter 6 summarizes the most important insights that we have gained through our work and offers future research directions.

Finally, in order to make the thesis self-contained, we present in the appendices topics that are relevant but not central to the thesis. In appendix A, we give elementary definitions from graph theory and define the notion of a cut. In appendix B, we survey problems and algorithms related to graph partitioning, which is a central part of Devine's load-balancing algorithm. Appendix C discusses non-uniform memory architecture (NUMA) machines and presents some experimental results about NUMA effects on the machine that we used to run the majority of our experiments. Appendix D gives a complete listing for the k-means program implemented for Nornir. Unlike the code snippets in the main body of the thesis, this listing shows aspects such as startup, shutdown and creation of processes and channels.

# 2. Background material

In this chapter, we survey existing tools for developing concurrent programs on two levels. Section 2.1 is dedicated to high-level issues of developing concurrent programs, i.e., abstractions, programming languages and frameworks. High-level support depends on low-level mechanisms for scheduling, communication and synchronization. These mechanisms, provided by the operating system, are described in section 2.2.

## 2.1. Parallel programming tools

A deep issue with parallel programming is that there is no universal way of converting a sequential program into a concurrent or distributed implementation. Once converted, a parallelized program often bears no resemblance to its sequential counter-part, but certain patterns have nevertheless been observed. We begin our survey by explaining two dichotomies: that between data- and task-parallelism, and that between shared-state and message passing concurrency. As KPNs are a task-parallel model, we then focus on existing tools for task-parallelism and relate them to the KPN model.

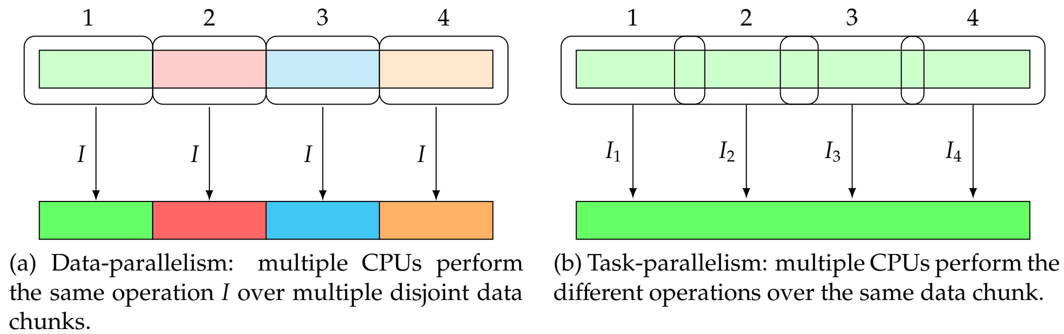
### 2.1.1. Dichotomies

The basis for the dichotomy between data and task parallelism is the question about *how* a single sequential task can be split into multiple (semi-)independent tasks. Similarly, the basis for the dichotomy between shared-state and message-passing parallelism is the question about *which* mechanisms tasks use to communicate with each other.

#### 2.1.1.1. Data vs. task parallelism

Data-parallel programs split the *input data*, while task-parallel programs split the *algorithm*. Of course, this classification is somewhat fuzzy, and many programs fall somewhere in between the two extremes. For example, it is easy to imagine splitting up a complex algorithm into multiple independent tasks (task-parallelism), each of which is internally data-parallel.

**Data parallelism** Data-parallel programs split the input data into many large chunks, usually as many as there are CPUs. Each chunk is assigned to a dedicated CPU, as shown in figure 2.1a, and all CPUs execute the same instruction sequence *I*. Furthermore, each CPU is usually also equipped with a set of wide registers containing multiple data




(a) Data-parallelism: multiple CPUs perform the same operation $I$ over multiple disjoint data chunks.

(b) Task-parallelism: multiple CPUs perform the different operations over the same data chunk.

Figure 2.1.: Data and task parallelism on 4 CPUs.

items and vector instructions for operating on those registers; this is the so-called SIMD architecture according to Flynn's taxonomy [33]. For example, newer x86 processors have been extended with SIMD aspects with the SSE instruction set and its successors, which have introduced 128-bit registers and many new instructions. Different SSE instructions interpret SSE registers differently: for example, a 128-bit register can be interpreted as containing 16 8-bit integers or 4 32-bit integers. A single SSE instruction operates on individual elements *in parallel*.

**Task parallelism** Task-parallel programs take the opposite approach, i.e., they split the algorithm into sequential and concurrently running tasks that explicitly distribute work among themselves. Each CPU executes a different instruction sequence, symbolized by labels $I_1 \ldots I_4$ in figure 2.1b. Unlike data-parallelism, where parallel operations are independent, tasks in task-parallel programs must communicate with each other in order to synchronize their activity. Communication may happen implicitly through shared-memory accesses (symbolized by overlapping CPU regions in figure 2.1b), or explicitly through message-passing.

**Trade-offs** The trade-offs between data-parallel and task-parallel approach to concurrency are manifested in two areas: ease of use and tool support.

We believe that task parallelism is easier to grasp because algorithms are most often expressed *imperatively*, i.e., as a sequence of steps over a *single* data set. Imperative description admits a natural decomposition into tasks by assigning each step to its own task. Data parallelism most often requires that the algorithm is transformed into a sequence of steps over *multiple* data sets. This transformation is algorithm-specific, and the result is often radically different from the sequential algorithm. Typical examples of data-parallel computations are the parallel prefix algorithm [34, 35] with its many applications, and MapReduce [8] computations, which we will discuss later in this thesis.

Despite being, in our opinion, conceptually harder to grasp and exploit, data parallelism has some advantages. First, synchronous execution and absence of data-sharing

between CPUs guarantee determinism and deadlock-freedom, and thus correctness of the overall program. Second, hardware can support it much more efficiently than task-parallelism, as witnessed by SIMD extensions to general-purpose CPUs as well as modern graphics processors (GPU) that can run *thousands* of threads simultaneously, provided that every thread executes the same code path. Each data-dependent branch stalls the GPU and slows down the execution. Third, it needs minimal support from the OS and minimal extensions to existing programming languages. In this respect, C and C++ compilers have been extended in several ways:

- By adding SIMD data-types as well as functions that generate the underlying CPU instructions.
- By extensions, such as CUDA [36] or OpenCL [37], which are used for programming highly data-parallel architectures such as GPUs.
- The OpenMP [38] standard is especially suited for loop-level parallelism, but later versions supports also task-level parallelism.

Despite being, in our opinion, conceptually easier to exploit, task-parallelism has also some drawbacks. First, it does not restrict communication in any way, so developers must be very careful to avoid problems like deadlocks, race-conditions and live-locks. The latter two classes of problems are notoriously difficult to reproduce and debug. Second, the abstractions and tools for exploiting task-parallelism are much more diverse and far less standardized, which can lead to interoperability problems when the developer wishes to use several tools in the same program.

#### 2.1.1.2. Shared-state vs. message-passing concurrency

In the vast majority of cases, the subtasks of a parallel program have some data-dependencies, i.e., task A will have to wait for the result of task B before it can continue with its own work. The waiting and communication of results can be accomplished in two ways: by sharing state between tasks, or by sending messages. Unlike the data-vs-task dichotomy, this one is much more crisp and it is rare to see an application that uses both shared-state and message-passing. A possible scenario would involve a distributed application that adapts to task placement: communication between tasks on the *same* machine would use shared memory, while communication between tasks on *different* machines would use more heavy-weight mechanisms, such as TCP/IP sockets.

**Shared-state concurrency** This is the most natural way of parallelizing *imperative* programs, i.e., programs that rely on *mutation* of values. The imperative paradigm encompasses procedural and object-oriented programming paradigms, which are today used for most software development projects. With shared-state concurrency, an imperative program is divided into tasks which are running within the same address space and which communicate by modifying shared data structures. The tasks must carefully use synchronization primitives, such as mutexes and condition variables, to coordinate accesses to shared data.

**Message-passing concurrency** Message-passing concurrency is common for implementing distributed programs, where individual tasks are running in separate address spaces. Communication is achieved by *explicit* calls to functions for sending and receiving messages; some programming languages support these operations through special syntax. Synchronization between subtasks is achieved as a part of communication, by using blocking send and receive, or by using barriers [39]. Communication in a message-passing application may be visualized by a *graph*, with vertices representing tasks and (directed) edges representing communication between the tasks. If a program can dynamically create or destroy processes, we say that the graph topology is *dynamic* (as opposed to *static* topology). Section 3.2 describes some well-known message-passing patterns.

**Trade-offs** The main advantage of shared-state concurrency are straightforward and low-overhead communication between tasks: changing a memory location makes the change visible to *all* tasks. This advantage is also its biggest drawback – it is easy to make a change to a shared data structure without protecting the access by a mutex or a similar mechanism. Not only are these mechanisms at a low leve of abstraction , but they are also non-deterministic: for example, many inexperienced developers are surprised to find out that threads waiting on a mutex are not woken up in FIFO order. Furthermore, the mutex state (locked or unlocked) has a *dynamic scope*, while modern programming languages use *static (lexical) scoping* for variables. The static program structure is therefore decoupled from its run-time behavior with respect to mutex state. Thus, mutexes are a rather low-level mechanism that is error-prone and hard to use even for experienced developers [6].

The main advantage of message-passing concurrency is that, unlike with shared-state concurrency, arbitrary interference between tasks is not possible, because all communication is explicit. Another advantage of message-passing concurrency is the availability of verification tools, such as SPIN [40], that can *prove* various assertions about program behavior. Nevertheless, message-passing APIs, such as MPI [41], are very general in that they offer facilities for direct point-to-point communication between tasks. The possibility of unrestricted asynchronous communication creates the possibility of interference through sending messages to unintended recipient processes.

A disadvantage of message-passing concurrency, at least when used on a single multi-core machine, are greater overheads of communication. These overheads generally include dynamic memory allocation and/or copying of message contents. Furthermore, broadcasting messages, which comes “for free” with shared-state concurrency, generally incurs overheads proportional with the number of tasks.

### 2.1.2. Tool support for task parallelism

In section 2.1.1.1, we have briefly discussed existing tools for exploiting data-parallel concurrency. We have also noted that tasks in a task-parallel program must coordinate their actions by altering shared-state or by exchanging messages. This dichotomy is used to structure the following discussion.

#### 2.1.2.1. Shared-state concurrency

We first review support for shared-state concurrency in major imperative programming languages in use today – C, C++, Java and C#. For each of these, we describe the support for concurrency in the core language and facilities available through libraries and language extensions. Then, we discuss two abstractions – software transactional memory (STM) [42] and futures [4] – which are today gaining ground in main-stream programming.

**C and C++** Even though they are two of the most widely used system programming languages, their current standards define *no* support for concurrency whatsoever, i.e., all support is delegated to libraries. The `volatile` keyword is meant to be used with variables that could be asynchronously changed, e.g., from interrupt handlers. However, the semantics of `volatile` variables is implementation-defined, and not all C and C++ implementations define it to cover memory visibility issues (see appendix C). The new C++ standard, still in the making at the time of writing this thesis, will support concurrency by defining a memory model, atomic types with necessary operations, and thread-local storage. Nevertheless, the bulk of concurrency support (threads, synchronization primitives, futures) will remain at the standard library level.

Thus, C and C++ currently support concurrency by platform-specific APIs (e.g., POSIX and Win32), platform-independent wrappers such as glib [43], boost [44], threading building blocks [45] or language extensions such as OpenMP [38] or $\mu$C++ [46]. The last three are also cross-platform and offer extensive feature sets.

**Threading building blocks** (TBB) is a C++ library that consists of several major components: parallelized loop algorithms, pipelines, concurrent data structures, mutexes, atomic operations, scalable memory allocator, and a task scheduler based on work-stealing [30]. The task scheduler is used to implement parallelized loops and pipelines. It implements a cooperative m:n scheduling model (see section 2.2.1) with tasks running *until completion*, and a load-balancing algorithm similar to work-stealing [47, 30]. The scheduler supports only fork and join operations, which order tasks in a strict parent-child relationship.

TBB lacks condition variables or other similar mechanisms; the only event that a task can wait on is termination of its child tasks, which makes it very cumbersome to implement other common parallelism patterns such as producer-consumer.[1] An advantage of limiting the scheduler to fork-join patterns is that tasks do not need their own stacks; instead they can borrow the stack from the thread in which they are currently executing.

**OpenMP**, mentioned in section 2.1.1.1, supports also explicit task-parallelism from version 3. It is an extensive standard that covers issues such as synchronization, atomic operations, and memory visibility issues, which we explain in appendix C. Its task model, however, supports only fork-join parallelism, so it has the same drawbacks as

[1] Indeed, the reference documentation [48] explicitly warns against using the producer-consumer pattern because it is not guaranteed that producer and consumer will be executing concurrently.

the TBB's scheduler. Since OpenMP is abstract API specification, the implementation details vary between platforms and compilers.

$\mu$**C++** [46] extends the C++ language with a number of concurrent programming concepts, such as coroutines [49], which are scheduled cooperatively, tasks, which run in parallel and are scheduled preemptively, and monitors [5]. A $\mu$C++ program is translated into regular C++ code and linked with a run-time library. Unlike TBB and OpenMP, which support only fork-join concurrency, $\mu$C++ supports general concurrency patterns.

**Java** The Java language supports shared-memory concurrency through core language and libraries. Unlike C and C++, the Java language defines a memory model with well-defined semantics of `volatile` variables, which helps with implementing light-weight concurrent algorithms. Task-parallelism is achieved with the help of the `Thread` class, the `Runnable` interface and the `synchronized` keyword. The `synchronized` keyword is used to define a monitor-like synchronization object [7] that can guard methods or individual statement blocks. The `synchronized` keyword is always associated with an object instance, which contains a mutex otherwise inaccessible to the application. The `synchronized` keyword causes that the mutex is locked upon entry to the block, and unlocked upon exit from the block. `wait`, `notify` and `notifyAll` methods, which can be used within `synchronized` blocks and methods, provide functionality of condition variables [5]. Additional concurrent data structures, classes supporting programming with futures, and other utility classes are provided in the `java.util.concurrent` package.

**C#** Similarly to Java, the C# language supports shared-memory concurrency through core language and libraries of the .NET platform. As Java, C# defines a memory model and introduces the `volatile` keyword with well-defined semantics. Unlike Java's `synchronized` keyword, C# uses the `lock` statement which can be used only on statement-level. A number of utility classes related to threading, among them also a `Monitor` class, are defined in `System.Threading` namespace of the standard library.

The C# language has been experimentally extended with concepts borrowed from join calculus [50]. The extension introduces asynchronous methods (methods are by default synchronous) and chords [51]. Asynchronous methods are queued for execution in another thread and return no result. Chords consist of a header, which specify patterns of method invocations containing at most one synchronous method, and a body, which defines an action for the corresponding pattern. The action is executed only after all methods in the header have finished execution. Completed asynchronous calls are implicitly queued at the chord as long as there is no matching header. When multiple queued calls of the same method match the same header, one of them is selected non-deterministically. Similarly, if several different chords become runnable, only one is selected for execution nondeterministically. Even a simple, *single-threaded* program may behave non-deterministically, as demonstrated in [51].

**Futures** Futures [4] are a construct that abstracts asynchronous function calls. A call to a "future function" immediately returns an encapsulation of the result ("future"), while the function continues to execute asynchronously. The returned future can be used to test whether the evaluation has finished, query the return value of the computation, or to cancel the computation. Querying the value returns immediately if the computation has finished; otherwise, the caller is blocked until the result becomes available. Once the computation has completed, its value is permanently remembered, i.e., further value queries will not re-execute the computation.

**Software transactional memory** Wtih software transactional memory (STM) [42], operations over shared data, that would usually be protected by locks, are replaced with code within an `atomic {}` block. The compiler and the run-time system work together to guarantee that all memory operations in the atomic block either succeed or fail. Memory accesses within `atomic` scope are replaced by calls into the run-time system which monitors consistency of memory accesses, with memory writes being only *tentative*. When an inconsistency is detected (e.g., when the run-time detects concurrent writes to same location from concurrent atomic blocks), all but one transaction are aborted, and their tentative modifications are discarded. When a transaction has successfully finished, its tentative modifications are *committed* and made globally visible to all threads.

STM is a complex technology, and questions have been raised about whether the advertised benefits of STM really do offset the additional complexity and performance costs that it introduces [52, 53]. Nevertheless, it is a trend that has captured the attention of some CPU-manufacturers, which have begun investigating methods for hardware-assisted STM [54].

#### 2.1.2.2. Message-passing concurrency

In this section, we first describe the message passing interface (MPI) [41], which is used in languages that have no support for message-passing concurrency in their core, such as C, C++, Java and C#. Then, we describe a number of languages that have been designed with message-passing concurrency in mind, and whose design is often based on some formal model of computation. We shall discuss Erlang [55], Occam and Occam-$\pi$ [56, 57], StreamIt [14] and SHIM [58], which are respectively based on the formalisms of actors [59], communicating sequential processes [60] and $\pi$-calculus [61], synchronous dataflow networks [62] and a restriction of Kahn process networks.

**MPI** The message passing interface (MPI) [41] has become the *de facto* standard for implementing distributed applications with C, C++ and Fortran; bindings are available also for Java and C#. The MPI standard is only an *interface specification*, with many implementations, both commercial and free. Version 1.2 of the MPI standard, which is most widely-used, provides process grouping, basic point-to-point communication as well as more complex collective operations (e.g., broadcast or reduce). Basic data types are automatically serialized for network transport, and the system can be extended to

support serialization of user-defined data types. Version 2.1 adds many new features, most significant being dynamic process management and parallel I/O. MPI thus defines *mechanisms*, but does not endorse any particular concurrency model.

**Erlang** Erlang [55] is a pure functional programming language based on the Ahga's actor model [59]. *Actors* are computational agents which have a behavior and which communicate with each other by sending messages via mailboxes. Sending a message is possible only if the sender know the receiver's address, which it can obtain either by creating the other actor, or by receiving its address in a message. Message arrival causes the actor's behavior to be executed, and its execution must always specify a new behavior, which will be used to process the next message. The only allowed side-effects of the actor's behavior are sending a finite number of messages to other actors, or creating a finite number of new actors.

Erlang's processes are restricted in their communication patterns in the same way as Agha's actors. However, unlike in the actor model, communication is order-preserving, i.e., arriving messages are queued instead of being "thrown in a mailbox". The receive operation is blocking and introduces a new concept, *pattern matching*, where a process gives a template consisting of values and variables. The receive operation dequeues the first message with matching values and fills in the variables in the template with the rest of message contents. Erlang programs are executed in virtual machine which very efficiently implements its process model: message passing, context switch and process creation take less than a microsecond, and each process initially consumes less than a kilobyte of space. Thus, it is possible to concurrently run millions of processes with little overhead [63].

**Occam** Occam [56] is an imperative programming language, developed by INMOS to support programming their transputer CPUs, which is based on Hoare's formalism of communicating sequential processes [60] (CSP). A CSP program consists of sequential processes running concurrently, which may communicate only synchronously through message send and receive operations. In other words, communication succeeds only when one process names another destination for output, *and* the other names the first one as the source for input. The CSP language supports neither recursion nor dynamic creation of processes, so the program code determines the upper bound on the number of concurrently running processes. The language also contains several non-deterministic constructs. Occam's strict compile-time checks ensure that the processes can share data only in the read-read mode, i.e., that no write is concurrent to another read or write.

The **occam-**$\pi$ language [57] extends occam with support for describing changing topologies and mobility of processes and data. The formal underpinnings of these extensions are borrowed from Milner's $\pi$-calculus [61], whose main constituents are names, which represent communication links, and processes, which are built from names according to the formal rules. The basic operations from which processes are built are send, receive, nondeterministic choice of communication among several alternatives, parallel composition and replication. As in the CSP formalism, communication between

processes is synchronous.

**StreamIt** The StreamIt [14] programming language is based on an extension of the synchronous dataflow [62] network model. Dataflow networks are a special case of KPNs, where activation of a process, also called an actor,[2] is controlled by a set of firing rules. Dataflow networks are obtained when the following additional restrictions are imposed on the Kahn model:

- The set of firing rules must be sequential, i.e., it must be possible to evaluate the rules in a predefined order by using only blocking reads.
- Each actor firing is functional, i.e., the set of output tokens is a pure function of the input tokens consumed in that firing.

Since actors are functional and execute atomically, i.e., they cannot be interrupted in the middle of their execution, there is no need to save state between two firings of an actor. Thus, the scheduler can schedule actor firings instead of full-fledged processes and avoid the costs of context switch and storage reserved to hold the actor's state (the stack). A dataflow network is *synchronous* if, for every actor and for every of its inputs and outputs, the number of consumed or produced tokens is constant and known *a priori*.

A StreamIt program is described by a graph consisting of computational blocks (filters) having a single input and output. Filters are combined into more complex structures by composite blocks, which provide pipeline, split-join and feedback loop structures. Filters must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous dataflow graph. This is exploited by the compiler which applies a number of optimizations [14] and produces a schedule before generating C++ or Java code. The generated code has the ability to run on a multi-core machine, or in a cluster, but the number of threads (or machines) is specified statically at compile-time. However, StreamIt is a special-purpose language, and there seems to be no way of using libraries of the target language, i.e., Java or C++.

**SHIM** SHIM [58] is a concurrent, asynchronous and deterministic model meant for developing embedded systems. The authors have taken a starting point in the KPN model, and deliberately restricted it to support only *synchronous* (rendezvous) communication. This choice eases scheduling, and programs are, by definition, always executable in finite space because synchronous communication does not need buffering of messages. In the same paper, Edwards et al. present algorithms for compiling programs written in a simplistic language, TinySHIM, to single-threaded C code, or to synchronous digital circuits.

[2] Not to be confused with Agha's actors.

### 2.1.3. Large-scale frameworks

Industrial actors have developed their own solutions for simplified distributed processing of large data quantities, such as Google's MapReduce [8], Yahoo's Pig latin programming language [9], Microsoft's Dryad [10], Cosmos and programming language Scope [11], and IBM's System S and programming language SPADE [13].

**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 expresiveness of System S is equivalent to that of KPNs. However, not much is known about these systems and their availability is limited.

**MapReduce** [8] 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. We shall describe MapReduce semantics in section 3.3.1 in detail, where we shall also discuss its drawbacks: rigid semantics and inability to model iterative algorithms; the latter being also a drawback of Dryad and Cosmos. A paper by Lämmel [64] extensively analyzes MapReduce, the related Sawzall language [65] and parallelization issues in the context of the Haskell [66] programming language.

Google's MapReduce implementation supports fault-tolerant distributed execution in clusters. Others have reimplemented MapReduce for clusters (Hadoop [27]), multi-core machines [26], the Cell BE architecture [28], and even for GPUs [29], which witnesses about its popularity.

**Pig latin** [9] is a language for performing ad-hoc queries over large data sets, where users specify their queries in an high-level language providing many features of SQL. Unlike SQL, which relies on query optimizers for efficient execution, Pig latin allows users to specify exactly *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. All Pig latin operators are also implementable as Dryad or System S operators, or as Kahn processes. Our experimental results (see section 5.1) indicate that compiling Pig latin programs into one of these more flexible frameworks would be advantageous for their performance.

## 2.2. Operating system facilities

In this section, we review OS mechanisms that are used for implementing concurrent and distributed programs. We focus on processes and threads, which are used for exploiting multiple CPUs, scheduling, communication and synchronization mechanisms. Our presentation synthesizes the material that can be found in text-books on operating systems [5] or in reference materials such as [67, 39, 68].

### 2.2.1. Processes and threads

The operating system's kernel handles scheduling of processes and threads. Each process or thread has control flow independent of any other, so the kernel can execute them concurrently on multiple CPUs, subject to scheduling policy.

**Processes** are independent of one another and do not share resources, such as address space and file descriptors. This increases reliability because a crash of one process will not affect any other. Reliability is reduced when processes use shared memory segments (because a buggy process can corrupt shared data) or synchronization mechanisms (because crashing while holding a semaphore or a mutex prevents progress of other processes). The main disadvantages of multi-process approach to concurrency, in comparison with threads, are higher bookkeeping and context switching overheads. For example, each process must have its own copy of page tables and file descriptor tables, and each context switch implies an expensive TLB flush.

**Threads** correspond to independent control flows within the *same* process, so they execute in the same address space and share all resources. Thus, communication through shared data is simple as a variable or a dynamically-allocated block of memory has the same address in every thread. Thus, pointer-based data structures can be shared between threads without problems, as long as the accesses are properly synchronized. Because threads share almost everything,[3] they consume less OS resources. Also, context switch between two threads of the same process incurs less overhead since threads are running in the same address space, so the TLB flush is unnecessary. The main disadvantage is reliability: a crash of one thread will crash the whole process, i.e., all threads.

There are two possible designs of the **threading model**, 1:1 and m:n models. In the **1:1 model**, each application thread has a dedicated kernel thread. The advantage of this approach is simplicity of implementation (a large amount of code in the kernel can be used to handle both processes and threads). The drawback is larger overhead of context switch, which always requires kernel intervention. In the **m:n model**, a user-mode library handles multiplexing of many threads over fewer kernel-level lightweight processes. Even though the m:n model can handle a large number of threads more efficiently than the 1:1 model (user-mode context switch is particularly cheap), major operating systems have abandoned it in favor of the 1:1 model because the latter is much simpler to implement correctly. The main implementation issues with the m:n model are handling of signals and blocking system calls.

### 2.2.2. Scheduling

The basic task of the OS scheduler is to distribute CPU time among processes or threads. The life-cycle of a process or a thread is depicted in figure 2.2. A process[4] is created in

[3] Some system aspects are nevertheless private to each thread. For example, on POSIX platforms, each thread has its own signal mask.

[4] For brevity, we will just write "process". The same applies to threads.

the ready state where it is eligible to be selected by the scheduler for dispatch on a CPU. Once dispatched, it enters the run state from which it can either:

- Yield or be preempted, entering the ready state.
- Exit and release resources associated with it.
- Block and enter the sleep state.

Once blocked, the process is not eligible for dispatch on a CPU until *another* process unblocks it, i.e., a process cannot unblock itself. After having been unblocked, the process enters the ready state and is again eligible for dispatch.

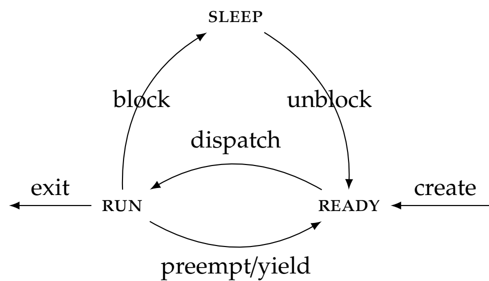



Figure 2.2.: Transitions between process and thread states. A sleeping process or thread can be unblocked only by another, running process.

Schedulers may be either preemptive or cooperative. When running under a *preemptive* scheduler, a process may be interrupted and descheduled (preempted) at any point of its execution. Under a *cooperative* scheduler, a process must voluntarily give up the CPU (yield) before another process or thread can be dispatched. While cooperative scheduling is simpler to implement and simpler to reason about, its disadvantage is that it cannot ensure fairness, so a run-away process that never yields can lock up the whole system.[5] Nevertheless, it was used in early multitasking operating systems such as Windows 3.

The *scheduling policy* is the algorithm which selects which of the ready processes and threads will run next; some typical examples are best effort, proportional-share and real-time. A *best effort* policy tries to give every process an equal time-slice, but may incorporate a number of heuristics that favors either interactive response time or batch throughput of applications. *Proportional-share* policies are governed by weights assigned to processes, with each process's time-slice being proportional to the process's weight. *Real-time* policies are governed by deadlines or priorities associated with each process. With a *deadline-driven scheduler*, a process declares its deadline and the scheduler guarantees that it will be dispatched in time to finish its work before the deadline expires. Deadline-driven schedulers employ admission control mechanisms to ensure that the CPU is not overbooked: they allow a new process to enter the system only if

[5]Hence the name: processes and threads must cooperate with each other to ensure global progress.

there is sufficient capacity to satisfy its scheduling requirements. With a *priority-based scheduler*, when a process of higher priority becomes runnable, it immediately preempts a process of lower priority, which is again dispatched when all higher-priority processes have blocked. Contention among processes of equal priority is usually resolved either by FIFO or round-robin policy with a fixed time-slice. Priority-based policies are widely available in POSIX operating systems such as Solaris and Linux, where the `sched_setscheduler` system call can be used to set scheduling policy and priority of a process. Setting a real-time policy requires administrator privileges because a run-away real-time process can completely lock up a system, requiring a reboot to recover.

Applications may give *affinity hints* to the scheduler to tell it on which subset of CPUs each thread will execute. Giving an affinity hint may improve performance on NUMA systems, but it may also cause load-imbalance, leading to worse performance.

### 2.2.3. Inter-process communication

Different processes (but not threads) are running in separate address spaces and are thus isolated from each other to a large degree.[6] Yet, a limited and controlled form of communication between processes can be advantageous, as witnessed by pipelines of the UNIX shell. POSIX offers many inter-process communication mechanisms, which we describe below.

#### 2.2.3.1. Pipes

A pipe is a *unidirectional* channel created by the `pipe` system call, which allocates a kernel buffer of fixed size and returns a pair of file descriptors, one for reading and the other for writing. Since the `pipe` system call does not create a name in the file-system for the pipe, i.e, the pipe is *anonymous*, it can be used only for communication between processes in the parent-child relationship. This is possible because file descriptors are inherited across `fork` calls.[7]

Pipes support only the byte stream abstraction, i.e., there are no message boundaries. To transfer data through the pipe, a process uses the returned file descriptors with `read` and `write` system calls in the usual way. Unlike files, whose size is known at all times, the operating system cannot know how much data will be transfered through the pipe. Thus, the writing process must explicitly signal the EOF condition to the reading process, which it does by closing the pipe with the `close` system-call.

A process that reads from an empty pipe or writes to a pipe with not enough empty space will block. However, POSIX mandates that writes of size less than `PIPE_BUF` bytes, which must be at least 512, are *atomic*, i.e., they must be performed without the data being interleaved with other writes. Writing to a closed pipe generates the `SIGPIPE` signal by default; if this signal is explicitly ignored, the `write` system call returns with the `EPIPE` failure code.

[6] Inadvertent interference may occur through the file-system.

[7] Though, some OS-es support sending file descriptors to unrelated processes through UNIX sockets.

*Named* pipes have identical behavior to anonymous pipes; the only difference is in the way they are created and opened. The mkfifo creates a named file of FIFO type in the file-system, but it does not allocate any kernel buffers. Once created, a named FIFO must be explicitly opened with the open system call, which will also allocate the FIFO's kernel buffer if the FIFO was previously unopened. The buffer is deallocated after the last process has closed the FIFO.

#### 2.2.3.2. Sockets

Sockets have been introduced in BSD UNIX family [68] as a general abstraction for communication between processes. Sockets are the only POSIX mechanism that enables communication between processes executing on different machines.

The socket system call takes domain, type and protocol parameters and creates a *bidirectional* communication channel. Socket domain selects the protocol family that will be used for communication. The most common domains in use today are PF_UNIX which selects data transport on the local machine, PF_INET which selects the IPv4 network protocol, and PF_INET6 which selects the IPv6 network protocol.

Socket type selects the transport semantics. The most common semantics today in use are SOCK_STREAM which selects reliable, order-preserving byte-based transmission, SOCK_DGRAM which selects unreliable transmission of datagrams, and SOCK_SEQPACKET which selects reliable, order-preserving transport of datagrams.

Even though the details of creating sockets and establishing communication are different from pipes, the basic semantics of write and read system calls is very similar. Additional, socket-specific system calls support the richer set of features offered by different protocols and are accessible through functions such as sendto, recvfrom, and setsockopt.

#### 2.2.3.3. Message queues

Similarly to named pipes, SYSV and POSIX message queues, also known as mailboxes, are used for communication between processes on the same machine. Although their basic semantics is somewhat similar to that of pipes, it is sufficiently different to introduce two new sets of functions. SYSV defines msgget, msgsnd, msgrcv and msgctl functions, while POSIX defines mq_open, mq_send, mq_recv, mq_close and a number of other functions.

The main difference between pipes or sockets and message queues is *persistence*. Message queues and messages and messages in them persist until they are explicitly destroyed or the system is rebooted. Specifically, closing a message queue will not destroy it, even if no other process holds it open. In addition, SYSV and POSIX message queues offer additional capabilities that are not available with pipes and sockets.

SYSV message queues support *message type*, which is an integer used to tag each message. The tag can be used to instruct the msgrcv function to fetch messages in several different ways.

- The first message in the queue, regardless of its type.

- The first message having type being either equal or unequal to the specified type.
- The first message with lowest type which is less than or equal to the specified type.

Message type can thus be used to implement message priorities or to *multiplex* several logical message queues over one physical queue.

POSIX message queues support *message priority*, which is an integer used to tag each message. The `mq_receive` does *not* take priority as an input argument; instead it returns the oldest message with the highest priority, optionally also returning the message priority. In addition, POSIX message queues can be configured to deliver an asynchronous notification (signal) when a message arrives on an empty message queue.

#### 2.2.3.4. Shared memory segments

The mechanisms described above make communication between processes possible through kernel mediation. However, kernel involvement incurs overheads in two ways:

- Through additional context switches and security checks if processes communicate by frequently sending small messages.
- Through data copying between user and kernel buffers if processes transfer large data volumes.

To avoid these overheads, processes can use shared-memory segments for communication. For both SYSV and POSIX shared memory, this entails two steps: creating or opening the segment by using `shmget` or `shmopen` functions, and *attaching* it to its own address space by using `shmat` or `mmap` functions.

Upon attach, the process may either let the kernel to choose the segment base address, or it may specify one itself. In the former case, the segment may be attached at a different address than in the other process(es), which is unsuitable for direct sharing of pointer-based data structures. In the latter case, the kernel will return an error if the segment cannot be attached at the specified address. This may happen, for example, when the mapping would overlap with an already allocated virtual address range.

When a process does not need the segment any longer, it *detaches* it by using `shmdt` or `munmap` functions. Like message queues, shared memory segments are also persistent, i.e., the memory is not freed when a segment is detached by the last process. Destroying a shared memory segment makes it inaccessible for futher open or attach operations, but resources associated with it are actually deallocated only when after all processes have detached it.

### 2.2.4. Process and thread synchronization

Synchronization primitives are used for two purposes: to enforce serial access to resources that cannot be safely accessed in parallel, and to provide a mechanism for waiting on events. On POSIX, both purposes can be achieved in several ways:

- By placing synchronization objects provided by POSIX threads (mutexes, condition variables, spinlocks) into a shared memory segment and configuring them to work across different processes.
- By using POSIX or SYSV semaphores, which can be used without setting up shared memory between processes.
- Blocking communication over message queues and pipes can also be used as a synchronization device.

In this section, we further discuss the advantages and disadvantages of the first two classes of synchronization mechanisms.

#### 2.2.4.1. Mutexes and condition variables

The disadvantage of using thread synchronization primitives between processes is that they can be left in an inconsistent state when a process crashes, which hinders correct functioning of other processes. Some operating systems, such as Solaris, provide *robust mutexes* as extension. When a process holding a robust mutex dies, the mutex is unlocked, and the next attempt to lock the mutex will acquire the mutex, but also return a special error code to inform the acquiring process that the previous owner has unexpectedly exited. This gives the new process an opportunity to repair the possibly inconsistent state left by the exited process.

#### 2.2.4.2. Semaphores

Unlike mutexes and condition variables, SYSV and POSIX semaphores do not require that processes share a memory region. Like message queues and shared memory segments, semaphores are also persistent. In addition, POSIX semaphores can be created in a shared memory area and used for synchronization between threads or processes. POSIX semaphores supports only the pure semaphore semantics [5]:

- A semaphore is initialized to a positive value.
- If incrementing the semaphore by one (`sem_post`) causes it to become greater than zero, a process or thread blocked in the semaphore will become unblocked and lock the semaphore.
- If a process or thread attempts to decrement by one (`sem_wait`) a zero-valued semaphore, it will be blocked until the semaphore becomes positive.

SYSV *semaphore sets* offer greater flexibility, which makes it possible to implement more complex synchronization patterns than with POSIX semaphores. Specifically, the `semop` function can perform the following functions over a semaphore set:

- Atomically incrementing or decrementing a semaphore in the set by a value greater than one.

- Waiting that a semaphore in the set becomes zero.
- Per-operation *undo flag*: when a process terminates, the operation on the semaphore will be automatically undone. Thus, if a process crashes while holding a semaphore, the automatic undo will allow progress of other processes.

One `semop` call can perform multiple operations over *several* semaphores in the same semaphore set, in which case all operations are performed *atomically*.

### 2.2.5. Deadlock detection

Operating systems have very limited support for deadlock detection – usually, only simple cases are detected, such as thread attempting to lock a mutex that it itself has already locked. Consequently, programs that need deadlock detection for more complex situations must implement it themselves. For example, a thread might only try to lock a mutex *without blocking*. If the mutex is unlocked, it will become locked and the operation will succeed. However, if the mutex is already taken, the OS will return an error code instead of blocking the thread. The thread has then an opportunity to check whether a deadlock condition exists and if so, act accordingly on it.

## 2.3. Summary

In this chapter, we have surveyed high-level and low-level aspects of concurrent programming.

In the first part of this chapter, we have explained the dichotomies between data- and task-parallelism and shared-state and message-passing concurrency. We have argued that task-parallelism, which encompasses also KPNs, is easier to grasp than data-parallelism because algorithms are expressed imperatively, so (groups of) individual steps can be extracted to different tasks.

We have then reviewed tools for supporting task-level parallelism in shared-state and message-passing concurrency models. On the shared-state concurrency side, we have described concurrency support in C, C++, Java and C# programming languages, and two programming techniques that are gaining ground in mainstream software development – futures and software transactional memory. None of the described mechanisms help with ensuring deterministic behavior, but we could have nevertheless used $\mu$C++ to *implement* a KPN framework. However, $\mu$C++ must be translated from an extended C++ language to standard C++, which introduces the risk of the translator not supporting all C++ constructs. Furthermore, in order to perform all our evaluations, we would have had to extend its rather large run-time support library with accounting and new scheduling mechanisms.

Our survey of message-passing systems with semantics similar to that of KPNs has revealed StreamIt [14] and SHIM [58], both of which adopt the process network paradigm. StreamIt *does* support feedback loops in the computation graph, but it is based on a synchronous dataflow network model, which result when further restrictions are imposed

on the KPN model. In addition, StreamIt is a new language which is translated to Java or C++, but provides no facilities to access libraries of the target language. SHIM [58] is a concurrent model of computation inspired by KPNs, but whose authors explicitly reject asynchronous communication, even though it has several advantages over synchronous communication, as argued by Agha [59]:

- Efficiency in execution by pipelining the actions to be performed.
- Synchronous communication can be defined in the framework of asynchronous communication.
- A process may be required to send messages to *itself*, which would lead to deadlock in the synchronous model. Worse, even mutual recursion is not possible with synchronous communication.

They argue for their choice with simpler scheduling and with the fact that such programs are executable in finite space, while the latter question is undecidable for the Kahn model.

Erlang [55], occam [56] and occam-pi [57] languages are respectively based on the Agha's actors [59], Hoare's communicating sequential processes [60] and Milner's $\pi$-calculus [61]. A common trait of these languages, as well as theories they are built of, is that they are non-deterministic.

In the second part of this chapter, we have described operating system support for implementing higher-level concurrency concepts. We have described the basics of scheduling, focusing on the difference between processes and threads, and between 1:1 and m:n scheduling models. We have then described OS support for communication and synchronization between processes and threads, as well as for deadlock detection. In the context of implementing a KPN run-time, the existing scheduling and synchronization mechanisms are functionally satisfactory. Communication mechanisms, on the other hand, have several problems: they are a scarce resource, there is no support for deadlock detection, and buffer capacity for some of them (e.g., pipes and POSIX message queues) cannot be enlarged, which, as we will explain in the following chapter, is a necessary feature for run-time deadlock resolution.

# 3. Modeling with Kahn process networks

In this chapter, we describe KPN semantics in detail and discuss general approaches to modeling parallel applications using KPNs. We relate KPNs to pipelines and MapReduce [8], and show that both can be implemented within the KPN framework. We also discuss the relative merits of KPN and MapReduce approaches to modeling parallel applications using two canonical MapReduce examples, word frequency and k-means, as case studies. Then, we demonstrate that KPNs can be simulated by iteratively executing MapReduce computations, thus establishing an isomorphism between KPNs and iterative MapReduce. Finally, we present existing KPN implementations and discuss their shortcomings.

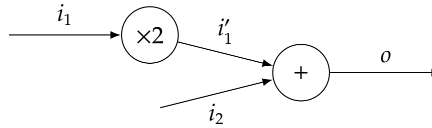
## 3.1. Semantics of Kahn process networks

In this section, we describe in detail the operational semantics of KPNs and its consequences, the most important of which is deterministic execution. We then discuss KPNs with dynamic topologies and limitations of deterministic parallelism. Finally, we discuss how to practically tackle the challenges posed by infinite channel capacities assumed by the theoretical model.

### 3.1.1. Operational semantics

A Kahn process network [12] (KPN) consists of sequential processes connected with unidirectional channels which together form a directed graph (see appendix A for a review of graph theory). Each process runs asynchronously and independently of other processes, and its state is inaccessible to other processes. Communication between processes is possible only by sending and receiving discrete messages over channels. Channels are unidirectional, reliable FIFO queues of *infinite capacity* that store messages. A process may have multiple input and output channels, but each channel is connected to *exactly two* processes – the sender on the one end of the channel and the receiver on the other end.

In **theory**, it message is always possible to send a message because channels have infinite capacities. A process may at any time attempt to receive a message from any of its input channels, but if the channel is empty, the process becomes *blocked* until a message arrives. Since testing for presence of messages on a channel is not allowed and blocking receive is uninterruptible, a process may remain indefinitely blocked in a receive operation. When all processes have become blocked, the KPN terminates. In **practice**, computers have only a limited memory, so the assumption about infinite




(a) Graphical representation of a KPN with input channels ($i_1, i_2$), external output channel ($o$) and an internal channel ($i'_1$).

```
void times2(void)
{
  while(true) {
    n = recv(i1);
    send(i1', 2*n);
  }
}
```

(b) Pseudo-code for the "×2" process.

```
void sum(void)
{
  while(true) {
    result = recv(i1') +
      recv(i2);
    send(o, result);
  }
}
```

(c) Pseudo-code for the "+" process.

Figure 3.1.: A KPN which calculates $o = 2i_1 + i_2$.

channel capacities is never fulfilled. A way of dealing with execution in constrained space is investigated in section 3.1.5.

#### 3.1.1.1. Example

Figure 3.1 illustrates a small KPN with two processes and four channels, which calculates the formula $o = 2i_1 + i_2$. Here, $i_1$ and $i_2$ are external inputs from the environment to the network, $o$ is an external output from the network to the environment, and $i'_1$ is an internal channel, i.e., output from the " × 2" process.[1] The "×2" process multiplies its input by 2 and sends it to the "+" process which sums values received at its inputs.

The pseudo-code for both processes is shown in figures 3.1b and 3.1c. The `times2` process executes an infinite loop in which it receives an integer from channel $i_1$, multiplies it by 2 and sends it to internal channel $i'_1$. Similarly, the `sum` process executes an infinite loop in which it receives the result of the `times2` process on channel $i'_1$ and another integer on channel $i_2$. Then, it sums the two values and sends the result to the $o$ channel.

#### 3.1.1.2. Determinism

The restrictions which operational semantics places on processes may be summarized as follows:

- A process may not access the state (code, data, program counter, etc.) of another process.

[1]By "environment" we mean the larger context in which the KPN is executing. External channels could, for example, represent the KP's direct use of file-system or network.

- The receive operation always blocks on an empty channel, and a blocked receive cannot be interrupted.
- It is not allowed to test for the presence of messages on a channel.

As a consequence of these restrictions, the KPN behavior is *deterministic*: the history of messages on channels is independent of the order in which ready processes are scheduled. In other words, given the same input, the network will produce the same sequence of message on all channels during every run.

#### 3.1.1.3. Scheduling

Since KPNs are deterministic, they can be scheduled by any *fair* scheduling algorithm. In this context, fairness is a rather weak requirement stating that execution of a ready process must not be indefinitely postponed. With an unfair scheduler, the output of a KPN will be correct, but it might be incomplete, i.e., the network might output fewer messages than under a fair scheduler [12].

For example, consider again the network in figure 3.1, and suppose that an *infinite* sequence of ones (1) arrives on both external input channels ($i_1$ and $i_2$). With fair scheduler, each of the KPs will be scheduled infinitely often, and an *infinite* sequence of messages with value 3 will be produced at the external output channel $o$. Now, suppose that the scheduler is *unfair* and that, for some reason, it will execute the "+" process only a *finite* number of times, for example 5. The output of the network will be a *finite* sequence, consisting of exactly 5 of messages with the value 3. Thus, the output messages are correct, but the output sequence is shorter than under a fair scheduler (obviously, $5 < \infty$).

### 3.1.2. Denotational semantics

As opposed to operational semantics, the denotational semantics describes a KPN with a set of equations, as we have done in the example KPN of figure 3.1. Kahn conjectured that the two semantics coincide if all processes in the network are *continuous*, a technical condition which is easily met in practice. Intuitively, a process is continuous if it does not consume infinite input before producing some output [12]. A formal treatment of this topic is given in [69, 62], and proofs of Kahn's conjecture can be found in [70, 71].

### 3.1.3. KPNs and coroutines

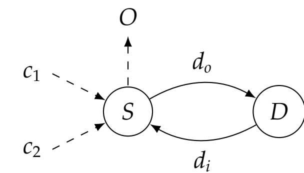
In another paper [72], Kahn developed a programming language based on processes communicating via unidirectional channels and having the same restrictions as KPNs, which ensure determinism. The language extends the process network model by defining a *reconfiguration statement*, which replaces a single node in the graph with several nodes, i.e., a subgraph. Such reconfiguration is possible provided that input and output channels of the subgraph can be appropriately spliced into the original graph.

He also describes two modes of running a process network: a coroutine mode and a parallel mode. In the *coroutine* mode, the execution is demand-driven with a single process driving the whole network. In this mode, control transfers happen on send/receive operations. Receive from an empty channel always yields to the sending process; send to an empty channel yields to the receiving process only if it is waiting for a message on that particular channel.

In the *parallel* mode, processes are running asynchronously, possibly on multiple CPUs, and Kahn notes that in this mode, processes may perform computations that are not needed for the final result. To counter this situation, he associates with each channel $C$ an integer $A_C$, called the anticipation coefficient. In the presence of $A_C$, the send operation will suspend the sending process if the target channel contains $A_C$, or more, messages.

### 3.1.4. Limitations of determinism

Since KPNs are deterministic, they are unsuitable for modeling *reactive systems*, i.e., systems that must respond to asynchronous events, such as interrupts from external devices or requests from multiple clients. Consider, for example, a disk scheduler with two clients, as shown in figure 3.2. The scheduler ($S$) reads requests from two clients along channels $c_1$ and $c_2$ in round-robin manner. After having read a request from a client, it forwards the request to the disk device ($D$) along channel $d_o$. Then, the scheduler waits for the completion message from the disk by reading from channel $d_i$. When the completion message arrives, the scheduler forwards it to the output channel $O$.




(a) Disk scheduler with two clients ($c_1$ and $c_2$).

```
void scheduler(void)
{
  for(i=0; i < 2; i = (i+1)%2) {
    send(d_o, recv(c[i]));
    send(out, recv(d_i));
  }
}
```

(b) Pseudo-code for the disk scheduler process.

Figure 3.2.: A KPN modeling a disk scheduler ($S$) with two clients and a single output ($O$) that handles replies for both clients.

The basic requirement on any scheduler is that all of its clients should eventually be serviced. In the KPN model, this is achievable by periodically reading a request from every client channel, as shown by the pseudo-code in figure 3.2b. However, because message receive is blocking and uninterruptible, any of the following events can severely degrade performance or even permanently block the scheduler and prevent it from serving clients:

```
void scheduler(void)
{
  bool eof[2] = {false, false};

  for(i=0; i < 2; i = (i+1)%2) {
    if(!eof[i]) {
      request = recv(c[i]);
      if(request != EOF) {
        send(d_o, request);
        send(out, recv(d_i));
      } else {
        eof[i] = true;
      }
    }
  }
}
```

Figure 3.3.: Disk scheduler implementing a protocol which ensures that the scheduler will not become permanently blocked when one of its clients sends only a finite number of requests. Compare the pseudo-code with that in figure 3.2.

- A client that issues a *finite* number of requests will permanently block the scheduler after having sent its last request.
- A client that issues infrequent requests will delay servicing of other clients, causing the disk to be idle much of the time.
- Hardware failure may cause disk operations take exceedingly long time or to never complete. In the latter case, the scheduler becomes permanently blocked on waiting for the completion message.

The first problem can be avoided by establishing a protocol between the scheduler and its clients, where a client sends a special EOF message after the last request. After having received an EOF message from a client, the scheduler will not try to receive a request from that client again, and thus avoid becoming permanently blocked; this behavior is depicted with the pseudo-code in figure 3.3.

The last two problems can be avoided only by introducing non-deterministic behavior. This is achievable in a structured way by extending the operational semantics with additional elements, such as non-deterministic merge, shown in figure 3.4. Unlike KPs, which can wait for input only on a *single* input channel at a time, the non-deterministic merge waits for input simultaneously on *all* of its inputs. As soon as a message arrives to an input, it is immediately forwarded to the output. Other ways of introducing non-determinism in the KPN semantics is by introducing m:1 channels (multiple senders to one receiver), or non-blocking receive.

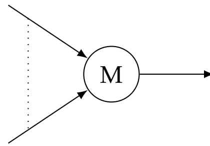
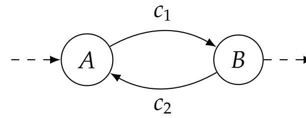



Figure 3.4.: The nondeterministic merge operator is a structured way of introducing non-determinism in a KPN. A message appearing on *any* input is immediately forwarded to the output.




(a) A part of a KPN where a local deadlock occurs.

```
void A(void)
{
  ...
  send(c1, data);
  x = recv(c2);
  ...
}
```

(b) Pseudo-code for the *A* process

```
void B(void)
{
  ...
  send(c2, data);
  x = recv(c1);
  ...
}
```

(c) Pseudo-code for the *B* process

Figure 3.5.: A local deadlock occurs when A and B execute the shown code with both $c_1$ and $c_2$ are full. Dashed arrows denote connections to the rest of the network.

### 3.1.5. A practical problem: execution in limited space

The assumption about infinite channel capacities is clearly unrealistic because any KPN implementation must run in a limited memory space. Real implementations assign finite capacities to channels and redefine the send operation to block if the target channel is full. With such modification of operational semantics, an *artificial deadlock* can occur at run-time. This is a situation where a subset of processes is blocked in a deadlock cycle, with at least one process being blocked on send. The deadlock is artificial because it would not occur with non-blocking send, i.e., infinite channel capacities.

Figure 3.5 shows an example of artificial deadlock in a part of a larger network. Processes $A$ and $B$ communicate with one another over channels $c_1$ and $c_2$ and with the rest of the network over channels denoted by the dashed arrows. If both processes happen to execute the code shown in the figure when both $c_1$ and $c_2$ are full, they will deadlock.

The cycle of deadlocked processes in figure 3.5 consists only of processes blocked on write. However, a cycle of processes in an artificial deadlock may consist of processes

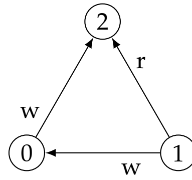



Figure 3.6.: A KPN with artificial deadlock cycle having one process blocked on *read*. The letters by channels denote that a process is blocked on read or write on the respective channel.

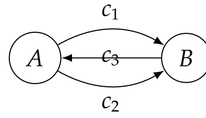
blocked on write *or read*, but with at least one process blocked on write,[2] as demonstrated by figure 3.6. Assuming all channels empty, the deadlock depicted in the figure can, for example, occur as follows:

- Process 2 starts by reading from channel $1 \to 2$, and immediately blocks, so it cannot read messages from channel $0 \to 2$, which would unblock 0.
- Process 1 starts by sending many messages to 0 and blocks because the channel has reached its capacity. Since 1 has blocked, it cannot send a message to 2, which would unblock it.
- Process 0 starts by sending many messages to 2 and blocks because the channel has reached its capacity. Likewise, it cannot read messages from 1, which would unblock it.

#### 3.1.5.1. Run-time deadlock detection and resolution

Since each KP is in its computational power equivalent to a Turing machine, it is impossible to statically compute channel capacities that are sufficiently large to ensure that artificial deadlock cannot occur at run-time. Thus, execution of arbitrary KPNs in limited space requires run-time detection and resolution of artificial deadlocks, which was first addressed by Parks [69]. His algorithm resolves the deadlock only when a *global* deadlock occurs, that is, when all processes in the network have become blocked. If at least one process is found to be blocked on write, the deadlock must be artificial, and it is resolved by increasing the capacity of the smallest full channel in the deadlock cycle. Geilen and Basten [73] have noticed that an artificial deadlock may be *local*, i.e., involve only a part of the network, as is the case in our first example shown in figure 3.5. Even though a local deadlock does not necessarily result in a global deadlock, it makes scheduling unfair because the deadlocked processes will never again run, so the KPN output will not be complete. They propose therefore a deadlock detection and resolution algorithm that handles also local deadlocks and resolves them in two steps:

[2]Recall that it is blocking write that causes artificial deadlock. If all processes have become blocked on read, that part of the network has terminated, i.e., the deadlock cannot be "resolved".




(a) The KPN.

```
void A(void)
{
  while(1) {
    send(c2, data);
    send(c2, data);
    send(c1, data);
    x = recv(c3);
  }
}
```

(b) Pseudo-code for the *A* process.

```
void B(void)
{
  while(1) {
    x = recv(c1);
    x = recv(c2);
    send(c3, data);
  }
}
```

(c) Pseudo-code for the *B* process.

Figure 3.7.: An effective KPN that is not executable in finite space.

1. Finding the deadlock cycle by traversing the chain of blocked processes and returning the full channel with smallest capacity.

2. Increasing that channel's capacity by one and unblocking the sending process.

### 3.1.5.2. Effective networks

A real KPN implementation cannot execute an *arbitrary* KPN in a finite space even if it implements a run-time deadlock detection and resolution algorithm. The simplest example of such KPN consists of two processes, *A* and *B*, where *A* sends messages to *B*, but *B never* reads them. In practice, it is reasonable to assume that each sent message is *eventually* read; such networks are called *effective* [73].

However, not even all effective networks are executable in finite space, which is demonstrated by the network shown in figure 3.7. *A* sends *two* messages over $c_2$, then it sends one message over $c_1$ to signal *B* that it may proceed, and then it waits for a message on $c_3$ before continuing to send data over $c_2$. *B* waits for a message on $c_1$, reads *one* message from $c_2$ and signals to *A* over $c_3$ that it may proceed. Thus, every message sent on $c_2$ is eventually consumed, but the number of messages on channel $c_2$ nevertheless grows without bound: whenever the two processes synchronize on $c_3$, two messages have been sent over $c_2$, and only one has been received, thus an increase of one message on $c_2$ per loop iteration.

As explained in section 3.1.1, the KPN model assumes that processes are running asynchronously. It is thus allowed that processes run at different speeds, and even that the *same* process can run at different speeds at different times. This can happen, for example, because processes are executed on different CPUs, or because the scheduler

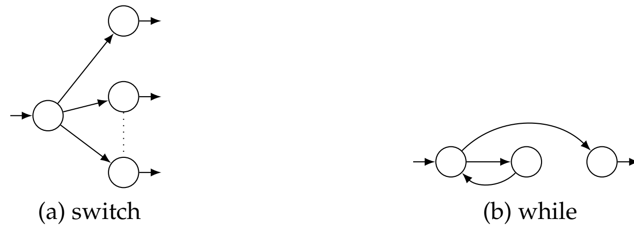



Figure 3.8.: Control structures realized as KPNs.

allocates a different time-slice to each process.[3] A valid example for the claim that not all effective networks are schedulable in finite space must account for the possibly that processes may run at unequal and non-constant speeds. Consequently, we need *two* channels from $A$ to $B$ to *ensure* that the number of messages on $c_2$ grows without bound, regardless of the relative speeds of the two processes.

## 3.2. Relation to known patterns

KPNs are more than just another way of expressing component composition; they represent a full-fledged graphical programming language which can express control structures such as multiple selection (see figure 3.8a), while loops (see figure 3.8b), direct and even indirect function calls. Because communication and execution are asynchronous, a large number of simple processes (fine granularity) may yield a greater degree of parallelism at run-time, but fine granularity also presupposes an efficient run-time environment that can support many processes.

However, modeling parallelism at very fine granularity is impractical for human beings (a KPN may be generated by a code-generator such as [16, 74]), so we shall focus on demonstrating how KPNs can model application-level parallelism at coarse granularity. In general, this entails three steps:

1. Identifying independent subtasks, which will become KPs, and their corresponding inputs and outputs.

2. Implementing KPs, possibly by "filling in the blanks" in off-the-shelf components, such as sort and n-way merge.

3. Determining the KPN topology, i.e., number of KPs and connections between them. The topology is largely dependendent on the previous steps.

We take a starting point in the well-known and established pipe abstraction and use it to define the parallel pipeline abstraction. Later, in section 3.3, we will show that MapReduce [8] is only a special case of a parallel pipeline.

[3]Frequency scaling on modern CPUs is yet another possible source of variation in execution speed.

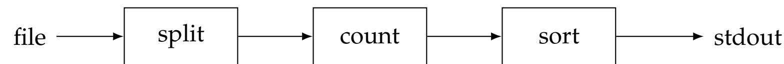



Figure 3.9.: Pipeline which computes word frequency in a set of files.

### 3.2.1. Pipelines

The pipe abstraction was introduced during the early development of the UNIX operating system [68], and its durability to present day witnesses about its versatility, simplicity and usefulness. UNIX pipes allow a complex task to be expressed as a composition of sequential programs which read their input from the standard input stream and write their output to the standard output stream. These two streams are usually connected to the user's interactive terminal, but they can be also redirected to a file or another stream. The standard output stream of one process can be connected to the standard input stream of another process by using the operating system object called a pipe, which we have described in section 2.2.3. Connecting the processes in this way creates a *pipeline*, an example of which is shown in figure 3.9. Besides simplicity, this paradigm also draws its power from its extensibility because users can create their own data-transformation programs and thus extend the set of built-in commands.

The pipeline in figure 3.9 shows how the task of finding most frequent words in a collection of documents can be expressed as a sequential composition of three stages performing smaller tasks: splitting the input into words, counting individual words, and sorting the word-frequency pairs by decreasing frequency. The split stage reads the input files, splits them into individual words, and outputs them, one word per line, to its standard output. The count stage reads the words from its standard input and maintains the number of occurrences for each word. When the count stage has exhausted its input, it will output the word-frequency pairs to its standard output. Finally, the sort stage reads those pairs, sorts them by frequency in decreasing order, and outputs the sorted list to its standard output.

The arrows between split, count and sort stages in figure 3.9 represent pipes, which transfer data between processes through memory buffers, i.e., without using temporary files. The flow control which is needed to execute such construction in finite space is managed by the operating system and is described in detail in section 2.2.3. The execution flow of this pipeline corresponds exactly to the execution flow of the same graph when viewed as a KPN. In other words, UNIX pipelines *are* KPNs, albeit with restricted topology, with UNIX processes corresponding to Kahn processes, and pipes corresponding to channels. However, as will be discussed later in this thesis, native UNIX mechanisms are very inefficient, and a large part of our work is dedicated to implementing better-performing mechanisms.

### 3.2.2. Parallel pipelines

The throughput of a pipeline is limited by the throughput of the slowest stage. When sufficient computing resources are available, the throughput can be increased in two

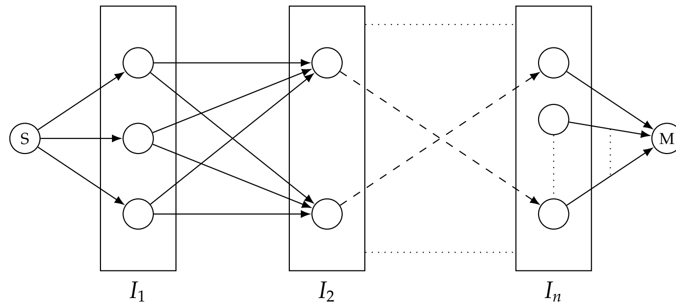



Figure 3.10.: Topology of a general parallel pipeline. The dashed arrows represent the same "butterfly" (all-to-all) connectivity between every pair of adjacent intermediate stages as between $I_1$ and $I_2$.

ways: by increasing the pipeline length, which also increases the latency of processing a single data item, or by parallelizing one or more pipeline stages, which does not sacrifice latency.

The latter approach yields the parallel pipeline construction, whose general topology within the KPN framework is shown in figure 3.10. We can identify three types of stages in the network: the splitter stage ($S$), the merge stage ($M$), and the intermediate processing stages ($I_1 \ldots I_n$). The splitter stage obtains the data, splits it into chunks and hands them off to the first processing stage ($I_1$). The intermediate processing stages perform the bulk of the work, with the last stage ($I_n$) producing the final output, which is partitioned. The merge stage finally coalesces the partitioned output into a single piece.

A processing stage is defined by the number of processes (e.g., $I_1$ in figure 3.10 has three processes) and the data transformation performed by each processes within the stage. The number of outputs and inputs of each process in adjacent stages is constrained so that every process in stage $I_j$ has as many outputs as there are processes in stage $I_{j+1}$ and, vice-versa, every proceess in stage $I_{j+1}$ has as many inputs as there are processes in stage $I_j$ (see connections between $I_1$ and $I_2$ in figure 3.10). Every process in stage $I_j$ is connected to every process in stage $I_{j+1}$, so the total number of channels between the stages is equal to the product of the number of processes in the two stages.

In section 3.3, we will encounter two concrete examples of a parallel pipeline. In both examples, a parallel stage sends messages to workers in the next stage according to the *value* of the data item in the message. Such partitioning is necessary both for load-balancing *and* correctness, and would not be possible without the all-to-all connectivity between stages. Nevertheless, the general topology of figure 3.10 may be simplified when allowed by the semantics of adjacent stages, as has been done in figure 3.11c.

### 3.2.3. Distributed parallel pipeline

In section 3.3.1, we will show that MapReduce is only a special case of a parallel pipeline. However, alongside of defining a parallel programming model, MapReduce

provides also distributed execution, fault-tolerance and distributed file-system with locality optimizations [75, 8].

These same elements can be used to provide distribution and fault-tolerance for parallel pipelines. As with MapReduce, distributed execution of a parallel pipeline would be coordinated by a master process which would start worker processes of individual stages, know about connections between them and observe their status. Worker processes of adjacent stages would communicate only by using the master process as a relay, and never directly with each other. In this scenario, the exchanged messages would typically be short, containing, for example, only locations of intermediate data files in the global filesystem. Thus, as in MapReduce, the master process has complete knowledge about remaining work of each worker and its status, so it can balance the load, restart failed workers or start backup tasks.

Since channels backed by files on a distributed filesystem have virtually unlimited capacity, worker processes will never block on send, so artifical deadlock cannot occur. Therefore, remote unblocking of a process and a distributed deadlock detection and resolution algorithm [22] are not necessary, which greatly simplifies the implementation.

## 3.3. Case studies

For a couple of years, Google's MapReduce [8] implementation has been the most cited approach for simple and high-level modeling of parallel and distributed computations. There exist third-party MapReduce implementations that run on clusters [27], multi-core machines [26], the Cell BE architecture [28], and even on graphics processors [29]. In this section, we show that MapReduce is only a special case of parallel pipeline, and is thus fully embeddable in the KPN framework.

Furthermore, we use two real-world applications, word frequency and k-means, to demonstrate that the rigid semantics of MapReduce incurs unnecessary overheads, and that these tasks can be implemented more efficiently in the KPN framework. We do this by implementing each application as a KPN in two different ways:

- The KPN-MR implementation builds a KPN that implements the exact MapReduce semantics (see section 3.3.1 and figure 3.11a).
- The KPN-FLEX implementation builds a KPN where the topology and behavior of processes is chosen to more naturally fit the given problem.

The results of our benchmarks, which are presented in section 5.1, show that KPN-FLEX implementations outperform the KPN-MR implementations. The KPN-FLEX implementations also outperform solutions implemented in Phoenix [26].

### 3.3.1. An analysis of MapReduce

In this section, we present the MapReduce semantics and show how it can be realized with a parallel pipeline in three straightforward, but different ways. Then we describe common data transformation stages (map, reduce, aggregate, sort) that the

three MapReduce realizations rely on. Finally, we analyze the benchmarks used by Phoenix [26]. Our analysis reveals that MapReduce solutions often perform more work than necessary, which reduces performance.

#### 3.3.1.1. Semantics

A MapReduce computation consists of two stages, Map and Reduce, which operate on a set of key-value pairs [8]. The Map stage applies a user-supplied function to every input key-value pair $(k, v)$ (record). The result of this application is a *set* of intermediate records $\{(k', v')\}$ which are passed to the Reduce stage. The Reduce stage can start after the Map stage has completely processed the input and generated the full output. The Reduce stage reads the output of the Map stage and sorts[4] the intermediate records on the intermediate key $k'$, which also brings together records with identical keys. Then it applies another user-supplied function on every group of values having the same key. The result of this function is the final output record for the group of values in question.

A MapReduce computation can be executed in parallel by having multiple parallel processes in each of the Map and Reduce stages. In this case, the Map stage is presented with an already partitioned data set, and it generates a partitioned output that is subsequently read by the processes in the Reduce stage. To ensure the correctness of the computation, the processes in the Map stage must always send intermediate key-value pairs with the same key to the same output partition.

#### 3.3.1.2. Realization with parallel pipeline

Figure 3.11 shows three ways in which a MapReduce computation can be expressed as a parallel pipeline which is, in turn, also a KPN. All three variants have the same semantics, but they may have different run-time performance, depending on data set size and distribution of input keys. Semantics of the parallel pipeline stages in the figure is as follows:

Map As in MapReduce, the map stage applies a user-supplied function to to every input key-value pair.

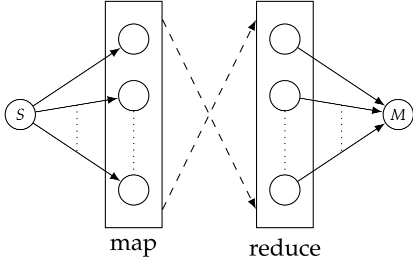
Reduce As in MapReduce, the reduce stage first reads the complete input from all channels into a single array, sorts the array by the intermediate key and applies a user-supplied reduce function to every group of records with identical keys.

Aggregate As the reduce stage, the aggregate stage reduces a list of key-value pairs with the same key to a single value. The main difference is that the aggregate can reduce its input *incrementally*. Unordered aggregate produces an unsorted output, while ordered aggregate produces output sorted by the key.[5]

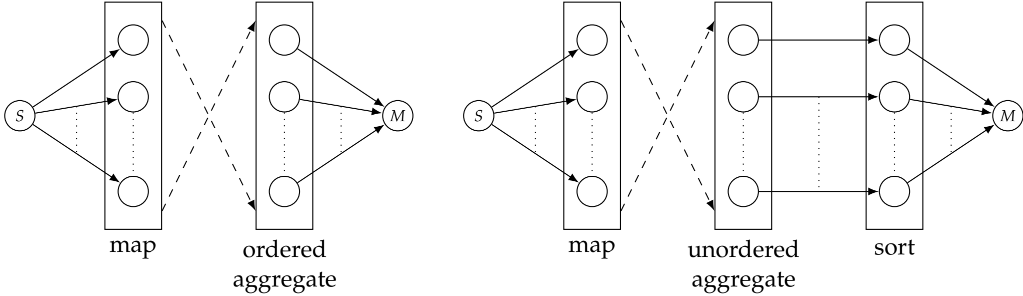
[4] The sorting step is the reason why Reduce must wait for Map completion – a sequence cannot be sorted until all of its elements are known.

[5] Unordered aggregate would typically be implemented with a hash table, and ordered aggregate with a balanced binary tree.

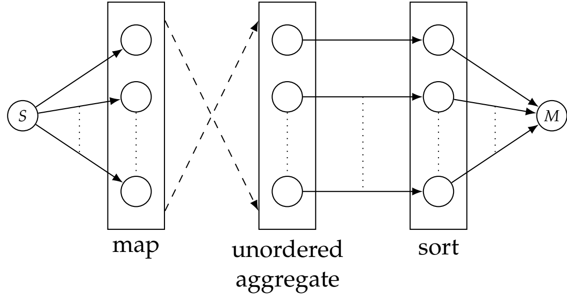
Sort The sort stage reads the complete input from its input channel, sorts it by the key using a user-supplied ordering relation, and outputs the sorted records.




(a) KPN duplicating operation of the original implementation described in [8].




(b) Implementation using ordered aggregate.




(c) Implementation using unordered aggregate and extra sort stage.

Figure 3.11.: MapReduce expressed as a parallel pipeline in three different ways, where every stage may have multiple workers. The dashed arrows represent all-to-all connectivity as detailed in figure 3.10.

When MapReduce is parallelized in one of the ways of figure 3.11, one must consider how data is distributed from the split stage $S$ to the map stage and from the map stage to the subsequent stage (reduce or (un)ordered aggregate). Defining the first distribution step is the developers' responsibility; the usual approach is to evenly divide the input data among the processes in the map stage. For the second distribution step, the choices are more limited: the map stage *must* ensure that intermediate records having the same key are sent to the same process in the subsequent stage. This ensures that each output record is computable by a single worker process.

One aims to achieve approximately even distribution of workload over processes in the subsequent stages. The usual approach is to send the intermediate record $(k', v')$ to the process determined by the formula $h(k') \bmod n$ where $h$ is some hash function and $n$ is the number of processes in the subsequent stage.

Examples of using MapReduce for solving real problems are illustrated in sections 3.3.2 and 3.3.3. In section 3.3.2 we will also see that split ($S$) and/or merge ($M$) stages of figure 3.11 can be omitted to avoid the overhead of merging the result of one MapReduce computation, just to split it again for the next MapReduce.

#### 3.3.1.3. A critique of MapReduce

We, and others [76], have noticed that the MapReduce semantics does not match all problems, which may lead to unnaturally expressed solutions and decreased performance.[6] We substantiate this claim with examination of the programs used to benchmark the Phoenix MapReduce implementation [26]. Our findings are summarized in table 3.1, where it can be seen that only 2 of the 8 benchmarks do not perform more work than necessary. We shall use two of the Phoenix benchmark programs, word frequency and K-Means, to illustrate that MapReduce is not an adequate implementation method, and that the flexibility offered by KPNs can give significant performance advantages.

| | |
|---|---|
| Word frequency | Two MR instances are needed because of unnecessary sort on the *wrong* key in the first instance. |
| Reverse index | Sort not needed. |
| Matrix multiply | Artificial keys, sort not needed. |
| String match | Sort not needed. |
| K-Means | Sort not needed. |
| PCA | Does not actually implement the full principal component analysis. Two MapReduce instantiations; Reduce stage is not needed in either. |
| Histogram | Map: counting color components in an image. Reduce: summing up. |
| Linear regression | Sort not needed. |

Table 3.1.: Phoenix bencmark programs.

### 3.3.2. Word frequency

As described in section 3.2.1, the word frequency program counts the number of occurrences of each word in a given text and outputs them sorted in the order of decreasing frequency. The solution to this problem requires file I/O, which we have provided through memory-mapped files. KPs can thus access the file contents by using pointers instead of by using I/O functions.

#### 3.3.2.1. KPN-MR solution

The KPN-MR solution closely follows the MapReduce solution presented in [26], which needs two consecutive MapReduce instantiations to solve this task. In addition to MapReduce computation stages, we have also introdced the split and merge process ($S$ and $M$ in figure 3.12) whose task is to get data in to and out from the computation stages.

[6]"When the only tool you have is a hammer, it is tempting to treat everything as if it were a nail." – Abraham Maslow in "The Psychology of Science."

The split process maps the input file into memory, divides it in chunks of equal size while taking care of word boundaries, and sends them to workers in the first computation stage. A file chunk is sent as a message containing a pair of pointers to the region of the file that the worker will process.

The first Map stage ("split" in figure 3.12a) outputs $(w, 1)$ pair for each word $w$ in the input file. The first Reduce stage ("sort, sum") sorts those records to obtain a list of pairs where identical words are grouped together. In other words, a list of the form

$$\ldots(A, 1)(A, 1)(A, 1)\ldots(V, 1)(V, 1)\ldots$$

is obtained. The user-supplied reduce function is addition with initial value 0. Thus, the Reduce stage simply sums the values (all of which are 1 in this case) over each group of identical words, and outputs a list of word-frequency pairs $(w, f_w)$ sorted by the key (word). For the above case, the output of the first MapReduce computation would be

$$\ldots(A, 3)\ldots(V, 2)\ldots$$

However, we are not interested in the word frequencies sorted *alphabetically*, but according to frequency in decreasing order. This is accomplished by another MapReduce application. The user-supplied function in the second Map stage ("reverse") reverses the key and value in the input, so that $(w, f_w)$ is mapped to $(f_w, w)$. The user-supplied function to the second Reduce stage ("sort") is identity, so the only effect of Reduce is to sort the records in decreasing key order. Thus, a MapReduce solution of the word frequency problem performs one unnecessary sort.

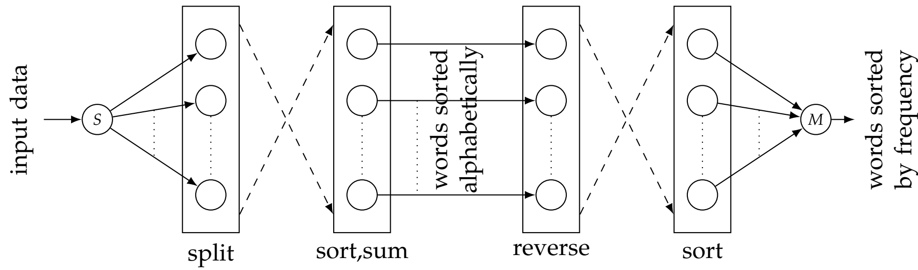
Finally, the merge process receives sorted results sets from individual workers in the last computation stage and merges them into a single sorted result set.

#### 3.3.2.2. KPN-FLEX solution

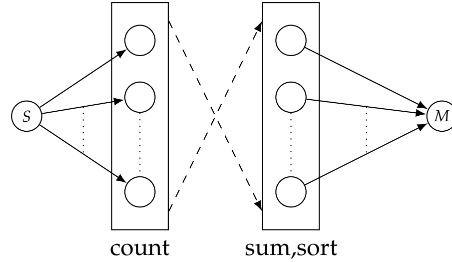
The KPN solution is a parallel pipeline consisting of two computation stages, as shown in figure 3.12b. A worker in the count stage counts individual words in its assigned file chunk. A worker in the sum-sort stage receives word frequencies from workers in the count stage and sorts them according to decreasing frequency. We also need the split and merge processes, which perform the same function as in the KPN-MR solution.

Figure 3.13 shows the code executed by the processes in the count stage.[7] The `behavior` method implements the `count` process. The process initially receives a text chunk which is represented as a `(start, end)` tuple of pointers pointing into a memory-mapped input file. The following `while` loop counts individual words in the text chunk by repeatedly calling the `parse_word` method. This method extracts the next word from the text chunk and updates the word's frequency stored in a hash table `counts_`. When all words from the chunk have been counted, *iterators* to each `(word, frequency)` pair

[7] The code snippet uses the boost tuples library [44]. `chunk` is a 2-tuple consisting of two `char*` fields that can be individually accessed by using the `get<>()` call. For example, `get<0>(chunk)` will access the first of the two constituent pointers.




(a) KPN that closely follows MapReduce operation. The number of processes in the second map stage is equal to the number of processes in the first reduce stage.




(b) KPN with optimized operation.

Figure 3.12.: KPNs for solving the word frequency problem.

in the hash table are sent to the sum-sort process. Finally, the `eof_all()` statement sets an EOF indication on all output channels. EOF signaling is our extension to the KPN framework, which does not alter its semantics, and which we shall describe in section 4.3.

Sending iterators is an optimization that avoids copying individual hash table elements. Load-balancing is accomplished by sending each pair to the output obtained by hashing the word and taking the result modulo the number of outputs. Since a hash function will always return the same result for a given word, this ensures that the same word and its partial frequency will be sent to the same process in the sum-sort stage. A process in that stage receives partial word frequencies from all count processes and maintains the total frequencies in another hash table. When all partial counts have been received, the hash table contains correct word frequency for each word. The process then copies the hash table into a vector, sorts it according to frequency in decreasing order and outputs the (word, frequency) pairs to the merge process, which performs a multi-way merge on the frequency field and produces the final result.

### 3.3.3. K-means

K-means is an algorithm used in data-mining applications. It is an iterative algorithm for partitioning a given set of points in multi-dimensional space into $k$ clusters, as exemplified in figure 3.14. In each iteration, the algorithm examines every input point and assigns it to the cluster whose center of gravity (centroid) is nearest to the point

```
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 3.13.: C++ code for the count stage of the word frequency program in KPN-FLEX implementation. The lines in bold font are the only necessary additions to the sequential version of the algorithm. For comparison, Google's MapReduce solution of the word frequency problem can be found in the appendix of [8].

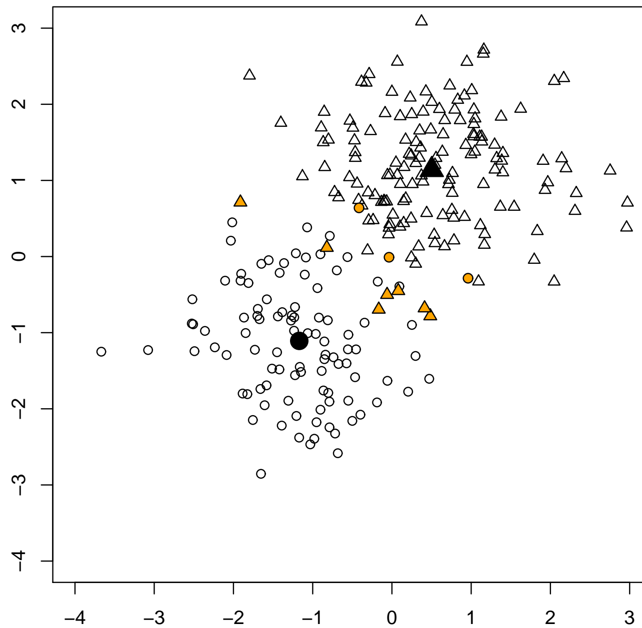


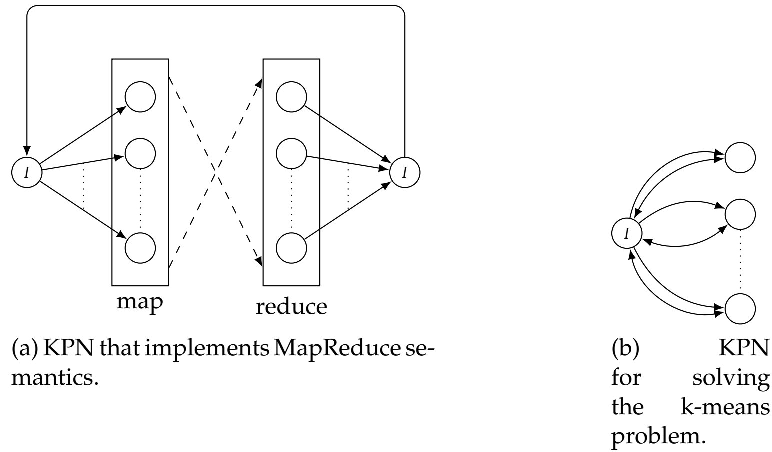

Figure 3.14.: Illustration of the k-means algorithm in a two-dimensional space. There are two point clusters with 100 and 150 points, plotted as circles and triangles, respectively. The large points filled with black represent the current centroid locations of each cluster. Points filled with orange color will be reassigned to the other cluster.

being examined. In more detail:

1. Guess, e.g., by random placement, the initial centroid locations.
2. For each point $p$ in the dataset, find the centroid $c_j$ which is nearest to $p$ and assign $p$ to cluster $j$. The location of $c_j$ is *not* recomputed in this step.
3. After all points have been assigned to (possibly new) clusters, recalculate new cluster centroids.
4. Repeat from step 2 as long as any of the centroids has moved by more than a prespecified threshold.

#### 3.3.3.1. KPN-MR solution

The KPN-MR solution closely follows the approach presented in [26]. We have also introduced a new process which initializes and iterates the computation. This is the $I$ process, which is represented with *two* nodes in figure 3.15 to make the layout less crowded.




(a) KPN that implements MapReduce semantics.

(b) KPN for solving the k-means problem.

Figure 3.15.: KPNs for solving the k-means problem.

The $I$ process consists of initialization and iteration phases. In the *initialization phase*, the $I$ process randomly generates $n$ points to be classified and $k$ points as the starting centroids. It partitions the point vector into as many approximately equal parts as there are worker processes in the next computation stage. Then, it sends a partition of the point set a pair of (start, end) iterators to each worker, followed by the $k$ centroids, each in its own message.

In the *iteration phase*, the $I$ process receives new centroid locations from the Reduce stage and compares them with the current locations. If at least one centroid has moved, it stores the new locations as the current locations and starts a new iteration by sending the new locations to the workers in the Map stage.

The map stage finds the nearest mean $c_i$ for each point $p$ and emits $(i, p)$ as the intermediate key-value pair. The Reduce stage computes new mean values from the itermediate key-value pairs. Since the MapReduce framework does not support modeling of iterative algorithms, the solution in [26] repeatedly runs MapReduce until the algorithm has converged. To avoid Nornir startup and shutdown costs, we have introduced the $I$ process described above.

#### 3.3.3.2. KPN-FLEX solution

Figure 3.15b illustrates a KPN implementing the k-means algorithm. Edges with double arrows represent *two* channels, one in each direction, with parallel channels carrying messages of different types.

As in the KPN-MR solution, we also use the iteration process $I$ to control the whole algorithm. The KPN starts running with the initialization phase of the $I$ process, which is identical to that of the KPN-MR solution. The iteration phase is different, and we shall describe it after having first described the algorithm executed by the worker KPs.

Figure 3.16 shows the code executed by the workers. A worker initially receives its part of the point set and then enters the main loop. In the main loop, it receives on its

input the current means and computes new cluster assignments for its part of the point set. As a worker assigns a point to a cluster, it also updates the count and partial sum of points belonging to that cluster. When a worker has processed all points, it sends the counts and partial sums for each cluster to the $I$ process.

In the *iteration phase*, the $I$ process receives from each worker the partial sum and the number of points for each cluster. Since mean is a linear operation, the $I$ process can now compute new location for all centroids. For a particular centroid $c_j$, its location is calculated by the formula $c_j = (\sum_i c_j^i)/(\sum_i n_j^i)$, where $c_j^i$ and $n_j^i$ are partial sums and counts for cluster $j$ sent by worker $i$. If at least one centroid has moved from its previous location, the $I$ process stores the new locations and begins another iteration of the algorithm by distributing the new locations to workers.

Even though workers modify state that has physically originated from the $I$ process, the $I$ process never inspects the original point set after having it distributed across workers; it only uses partial sums of the centroids that arrive on the other channel. Thus, determinism is preserved. The process of full ownership transfer could be enforced by using a class such as `auto_ptr`, which is a part of the C++ standard library.

In figure 3.16, the additional code necessary to transform the sequential algorithm into a Kahn process is shown in bold text. Note how this code resembles programming with ordinary files: a process reads its input from input ports (files opened for reading) one item at a time, does some processing, writes results to its output ports (files opened for writing), and exits when EOF is detected on some input port (file).

## 3.4. Executing KPNs by iterative MapReduce

As mentioned in section 3.2.3, MapReduce not only defines a programming model, but also handles distributed execution over a cluster of machines, fault-tolerance and load-balancing. Nornir, on the other hand, runs only on a single, multi-core machine. Since we do not know of a KPN run-time that supports distributed execution and fault-tolerance, we have derived two constructions, neither of which we have implemented, that use an *iterative* MapReduce computation to execute an arbitrary KPN.

In our descriptions, we omit the details of local deadlock detection in order to simplify the presentations of our constructions. Also, it is not unrealistic to treat channels as if they had infinite capacity, especially in our impure construction, since a distributed MapReduce computation often uses a distributed file-system.

We present two constructions in which a MapReduce computation can execute an arbitrary KPN:

- The *pure* construction exploits only the abstract MapReduce semantics, described in section 3.3.1, without assuming any particular implementation details.
- The *impure* construction assumes that all Map and Reduce worker processes have access to a global distributed file-system.

Both constructions share the common main idea: *to handle KP code as data*. The user-supplied map and reduce functions host an embedded interpreter for the KP implemen-

```
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 3.16.: C++ code for the worker processes of the k-means program in KPN implementation. The lines in bold are the only necessary additions to the sequential algorithm. When the *I* process sees that the algorithm has converged, it sets EOF status on its channels, thus making other processes exit.

| Field | Description |
|---|---|
| `pid` | Globally unique ID for the KP. |
| `chnid` | Mapping from *local* channel IDs, which are indices into the `chnid` array, to global channel IDs. |
| `chnbuf` | An array of references to buffers holding messages for each input channel. |
| `chnpid` | An array of receiving process IDs for each output channel. |
| `seqno` | KP's current sequence number. |
| `blocked` | Flag signaling that the KP is blocked on receive. |
| `send` | Implementation of the send primitive. |
| `recv` | Implementation of the receive primitive. |
| `data` | Context-dependent, see text. |

Table 3.2.: Components of the KP state that are directly accessed by the host function. The `chnbuf` and `chnpid` arrays are indexed by channel IDs local to the KP.

tation language. They receive the frozen interpreter's state (which includes KP's code and data) and contents of input channels, and execute the KP in the hosted interpreter. Interpreting the KP code will change its state, so the output of the Map and/or Reduce stages is a modified KP state and the set of new messages sent over the KP's outgoing channels. Table 3.2 summarizes the KP state that is accessed by the Map and Reduce functions.

The challenge is to cast the above high-level idea into a key-value paradigm enforced by MapReduce. Our construction assumes that the implementation language for KPs supports the following features:

- Coroutines [49] that can exit to the embedded interpreter while preserving the complete call-stack.
- Save and restore of the interpreter's state (which includes KP code, data, and channel contents; see table 3.2 for details).
- The interpreter supports reading and defining of variables and functions directly from the host function.

These assumptions are fulfilled by, for example, the Javascript language and its Rhino [77] implementation, which is written in pure Java. It would thus fit nicely with the Hadoop MapReduce implementation [27], which is open-source and also written in Java.

Both constructions also need a controller process, which creates the KPs and channels between them, fills channel buffers with initial contents and creates KP states that will be read by the first MapReduce iteration. Then, it enters a loop in which it repeatedly executes a single MapReduce pass, where the result of the Reduce stage of the previous pass is directly fed to the input of the Map stage in the next pass. At the end of each iteration, the controller checks whether all KPs have become blocked on read, and if so, terminates the execution. The controller process would also be the natural point for performing detection and resolution of local deadlock.

### 3.4.1. Pure solution

The pure solution uses solely the abstract MapReduce semantics, and does not rely on any implementation details, such as the existence of external resources accessible to the map and reduce functions, such as distributed file-system.

#### 3.4.1.1. Keys and values

The first step in casting our idea into the MapReduce key-value paradigm is to assign unique identifiers, e.g., positive integers, to processes and channels. Messages also need to be tagged with sequence numbers to ensure that they are processed in the sending order. The complete contents of the key-value records is thus

```
(pid, chnid, seqno, data)
```

where the first three fields are copied from the KP's state (see table 3.2). The `data` and `seqno` fields are interpreted according to the value of the `chnid` field:

- When `chnid == 0`, the `data` field contains the complete code and data belonging to the KP having the given `pid`. The `seqno` field contains the next sequence number that will be used for messages sent by this KP.
- When `chnid > 0`, the record represents a message whose *receiving* process is identified by `pid`. In this case, the `chnid` field is the global ID of the target channel, `data` is the actual message contents, and `seqno` is the message's sequence number which ensures correct ordering of messages on the channel.

#### 3.4.1.2. Map stage

In this construction, the Map stage does not perform any computations. Its only function is to distribute records across workers in the Reduce stage, ensuring that records with the same `pid` field are sent to the same worker.

#### 3.4.1.3. Reduce stage

The Reduce stage will first sort all input records according to the key, `pid`, thus grouping together all records belonging to the same KP. The user-supplied reduce function is then invoked with the list[8] containing elements having the following structure:

```
(pid, chnid, seqno, data)
```

All elements in the list have the same `pid` value, i.e., they belong to the same KP. The reduce function performs the following steps:

[8]"List" here does not presuppose any particular data structure; it merely denotes a "container whose elements are are arranged in a strict linear order" [78], such as linked list or array.

1. Lexicographically sorts the list of quadruples using `chnid` and `seqno` as the key. This brings together all messages destined for the same channel, and arranges them in the sending order.

2. Creates an interpreter instance that will execute the KP in question.

3. Loads the complete KP state into the interpreter. The KP state is encoded with `chnid==0`, so it will be found at the *front* of the list.

4. Appends messages in the rest of the list to their destination channel buffers in the KP state.

5. Defines `send` and `recv` functions as described below.

6. Executes the KP in the interpreter until the KP yields.

7. Emits the KP state as the last output record with `chnid` set to 0, `seqno` set to the KP's current sequence number and data filled in with the KP's state. Existing messages in channel buffers are *not* converted to individual records; they remain encapsulated in the KP data.

One might think that the sorting in step 1 could be avoided by letting the Reduce stage lexicographically sort its input using the triple `(pid, chnid, seqno)` as the key. With this approach, each record belonging to the same KP would be processed by a separate invocation of the reduce function. Since the state of the reduce function is not preserved across invocations, it would be impossible to restore the full KP state in steps 3 and 4 and execute it afterwards. However, step 1 *can* be omitted *if* the MapReduce implementation fulfills two conditions:

- Records output by the Reduce stage are not reordered.

- The sorting algorithm employed by the Reduce stage is stable.

A stable sorting algorithm preserves the relative order of elements with equivalent keys, so messages destined for the same channel will already be in the sending order. Since the KP state is output as the last record in step 7, step 3 will find the KP state at the *back* of the list.

The `send(lchnid, msg)` statement sends data to the channel identified by the local channel identifier, which is an index into the KP's table of global channel IDs (see table 3.2). The host will define the `send` function to perform the following steps:

1. Emit the output record[9] with `chnid` set to the global channel ID, `pid` set to the receiving KP ID, `seqno` set to the KP's current sequence number, and `data` filled in with the message data.

2. Increment `seqno` by 1.

[9]MapReduce allows generating multiple output records with the same key.

3. If the KP has sent more than a preset number of messages in this execution, yield (*not* block).

The last step ensures fairness towards other KPs, in case the same Reduce process will execute several KPs.

The `msg = recv(lchnid)` statement receives data from the channel identified by the local channel ID. Since the reduce function has filled the channel buffers before starting the KP, the performed steps are straightforward:

1. If there are messages in the channel buffer, set the `blocked` flag to false, remove message at the front of the specified channel and return it to the caller.

2. Set the `blocked` flag to true and yield.

3. When resumed, go to step 1.

### 3.4.2. Impure solution

In the impure variant, we assume that the user-supplied map and reduce functions have access to a global distributed file-system, which will be used to buffer messages. In this construction, the Map stage will allow processes processes to only *send* a message, with the limit on the maximum number of sent messages per execution. Similarly, the Reduce stage will allow processes to only *receive* a message. Such alternate scheduling of KPs is clearly fair, so it guarantees that the KPN will be output-complete. It is also advantageous because channel reads and writes are *never* concurrent, so the distributed file-system does not need to implement special support for concurrent atomic reads and writes.

#### 3.4.2.1. Keys and values

To map the problem to the key-value paradigm of a MapReduce computation, we define the record type that will be processed by the Map and Reduce stages:

```
(pid, state)
```

where `pid` is the unique KP ID, and `state` is the complete KP state, shown in table 3.2, that is used to resume its execution. The `data` field contains the complete code and data of the KP identified by `pid`.

#### 3.4.2.2. Map stage

The Map stage allows the processes to only send messages. As soon as a process attempts to receive a message, it will immediately yield. The user-supplied map function receives the record defined above and proceeds with the following steps:

1. Defines the `send` function in the KP state to append the message to the target channel file.

2. Defines the `recv` function in the KP state to immediately yield control to the host program.
3. Restores the context of the KP interpreter, opens the output channel files using information from the KP state (see table 3.2), and executes the KP until it yields.
4. Closes the output channel files and outputs the KP's state to the Reduce stage.

The send operation of step 1 shall yield (*not* block!) if the process sends more than a preset number of messages without an intervening receive. This ensures fairness towards other KPs, in case there are fewer Map workers than ready KPs.

#### 3.4.2.3. Reduce stage

The Reduce stage allows processes to only receive messages. As soon as a process attempts to send a message, it will immediately yield. The user-supplied reduce function receives the process state as defined above and proceeds with the following steps:

1. Defines the `send` function in the KP state to immediately yield control to the host program.
2. Defines the `recv` function in the KP state to read the next message from the channel file.
3. Restores the context of the KP interpreter, opens the input channel files using information from the KP state (see table 3.2), seeks each file to the record indicated by the `seqno` field, and executes the KP until it yields.
4. Closes the input channel files. If the files have grown too large, they are truncated at the beginning and `seqno` is set to 0.
5. Outputs the KP's state as the final result of the MapReduce computation. The result can be immediately fed to the input of the Map stage of the next MapReduce run.

The `recv` operation sets the `blocked` flag and yields control if it encounters EOF on the channel file. Otherwise, it sets the `blocked` flag to false and updates the `seqno` field to point to the next record in the channel. Unlike with `send`, a KP does not need to yield after receiving a preset number of messages without an intervening send. Any physical channel realization will contain only a *finite* number of messages, so a KP *must* eventually block on an empty channel. Fairness is thus automatically ensured.

## 3.5. Existing KPN implementations

**Yapi** [20], developed in C++, is aimed towards designing signal processing applications. YAPI is not a pure KPN implementation since it extends the semantics by introducing the channel selection operator, which introduces nondeterminism. With channel selection,

a process may block on a *set* of channels until an action can be completed without blocking on at least one of them. Furthermore, the user manual does not indicate that detection and resolution of artificial deadlock is implemented.

The paper [20] gives very few implementation details, but the code can be downloaded from `http://y-api.sourceforge.net/`. The code cannot be compiled without small fixes, and it is unstable, i.e., it crashes on Linux even though Linux is one of supported platforms. Further investigation revealed that Yapi implements m:n scheduler, and uses `setjmp` and `longjmp` functions to perform context switching, which is not guaranteed to work.[10] Lastly, the code-base is too large for easy experimentation, which is one of our requirements: YAPI has 120kB of source code, whereas our implementation has only 50kB, which includes platform abstraction layer, artificial deadlock detection and resolution, and two scheduling mechanisms (1:1 and m:n).

The **process network frameworks** developed by Evans et al. [22, 21][11] are another implementation of KPNs in C++. One of their implementations supports also distributed execution and deadlock detection and resolution algorithm [22]. However, their code supports only the 1:1 scheduling model, and it would require substantial extensions to introduce support for m:n scheduling. In fact, the m:n scheduler and the three scheduling policies, described in the next chapter, comprise $\sim 50\%$ of the complete Nornir's code-base.

**Ptolemy II** [19] is a simulation environment for prototyping and experimentation with heterogeneous systems. It provides several computational models, among them also KPNs, and supports their intermixing. A GUI for easy prototyping is also developed. KPNs in Ptolemy II are built on Java threads and monitors. The amount of code that the JVM consists of would make it prohibitively difficult to experiment with low-level mechanisms, such as lightweight tasks.

The **Sesame** project [15] features an event-driven *simulator* of embedded systems, which employs KPNs for application modeling and simulation. By collecting detailed traces during a simulation run, it assists in matching an application to the given hardware platform, or vice-versa. Since an event-driven simulation significantly slows down execution, Sesame is not suitable for executing KPNs where performance is important.

## 3.6. Summary

In this chapter, we have examined modeling of parallel applications with KPNs. We have demonstrated that a pipeline is a special case of KPN, and we have developed a general KPN topology that implements a *parallel* pipeline. Each stage of a parallel pipeline consists of several processes running in parallel, which increases the pipeline throughput without sacrificing latency. We have also shown that the widely-used MapReduce model of parallel computation is a special case of a parallel pipeline, which

[10]New linux C libraries make the `jmp_buf` structure truly opaque by xor-ing the contents with a per-thread random constant. This could be probably disabled by the undocumented `LD_POINTER_GUARD` environment variable on some Linux versions.

[11]The code is available at `http://users.ece.utexas.edu/~bevans/projects/pn/`

consists of two stages whose behavior is prespecified to a great extent.

Because of its rigid semantics, MapReduce performs more work than necessary on many problems. To see whether the modeling freedom allowed by KPNs brings any real performance benefits, we have implemented the word frequency and k-means applications within the KPN framework in two ways: the KPN-MR variant, whose topology and operation closely follows the MapReduce abstract semantics described in section 3.3.1, and the KPN-FLEX variant whose topology and operation is adapted to each problem. Our experimental results, presented in detail in section 5.1, show that the KPN-FLEX variant outperforms not only the KPN-MR variant, but it outpeforms also their implementations in Phoenix.

Nornir runs only on a single, multi-core machine, while MapReduce implementations are distributed over a cluster of machines, and also implement fault-tolerance. Since we do not know of a KPN run-time that supports distributed execution and fault-tolerance, we have derived two constructions, neither of which we have implemented, that use an *iterative* MapReduce computation to execute an arbitrary KPN. Thus, a MapReduce implementation can be used as an execution engine to provide distribution and fault-tolerance for KPNs, until a true KPN run-time with these properties is developed.

Finally, we have reviewed several most complete existing KPN implementations: Yapi [20], the process network frameworks by Evans et al. [22, 21], Ptolemy II [19] and the Sesame project [15]. We have found that none of them completely satisfies the requirements stated in section 1.2, so we had to implement Nornir, which design and implementation is the topic of the next chapter.

# 4. Nornir implementation

Nornir is our configurable run-time environment for executing KPNs, implemented in C++. Although it is primarily targeted towards POSIX environments, it also executes on newer Windows operating systems, but only in a subset of the configurations supported under POSIX. Its design is greatly influenced by our requirements: support for multiprocessing, easy experimentation with implementation techniques, support for artificial deadlock detection, collection of run-time statistics and support for large networks. The main components of Nornir are KP management and scheduling, load-balancing, message-passing, deadlock detection and collection of detailed statistics (accounting). Such decomposition is a natural consequence of the KPN model and our requirements. The implementation of each component can be changed independently of any other component, so it is easy to evaluate performance effects of each component individually.

Nornir implements preemptive 1:1 and cooperative m:n scheduling models, which have been described in the context of operating systems in section 2.2.2. With 1:1 model, one OS thread is created for each KP. With m:n model, many KPs are time-multiplexed over one OS thread, which we also call *runner*. In this chapter, we focus on the m:n model since it is more complicated to implement (see figure 4.1).

Our prsentation proceeds on two parallel levels. On the higher level, we describe data structures and algorithms that we have used in our implementation. On the lower-level, our descriptions are also meant to expose important implementation details and to be a guide to the Nornir source code.

## 4.1. Process management and scheduling

The process management and scheduling component (scheduler) orchestrates the whole life-cycle of a KPN: the startup phase, the running phase and the shutdown phase. Its role and operation is similar to that of regular OS schedulers introduced in section 2.2.2. In the startup phase, the scheduler invokes the user-supplied *root process* which creates KPs and channels. While the KPN is running, the scheduler balances the load by choosing which KPs will be executed, when, and on which CPU. Also, all KP block and unblock operations go through the scheduler, so it is able to keep track of the number of ready processes in the KPN and shut down the KPN when this number reaches zero. The scheduler is also a natural component for performing run-time deadlock detection and resolution because an artificial deadlock can occur only after a KP blocks, an event of which the scheduler is always "aware" of. In this section, we present the scheduler's main control-flow, i.e., startup, shutdown and management of the KP's life-cycle which

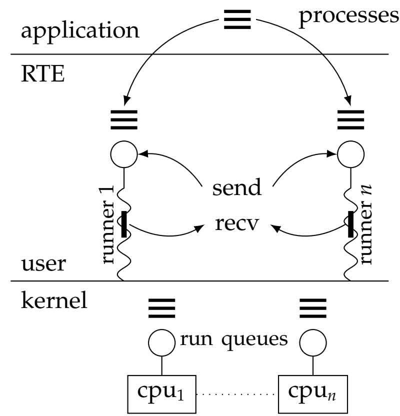



Figure 4.1.: KP scheduling: each runner has its own private run queue (small circles) containing ready KPs (small black squares), and is executing at most one KP at any given time. Runners are bound to different CPUs, so they never compete with each other. Nevertheless, they compete with other threads and processes.

is the same as that of regular processes and threads (see figure 2.2). The other two aspects related to scheduler's operation, load-balancing and deadlock-detection, are described in sections 4.2 and 4.4 respectively.

The m:n scheduler is cooperative, which means that a KP runs uninterrupted until it exits or encounters a blocking point. In our implementation, the only potentially blocking points are currently only message send and receive operations. Since KPs may use all UNIX system calls, they can perform I/O in the usual way. Even though I/O operations may block, they will not cause the calling KP to be block or yield. Instead, the runner executing the KP will become blocked and thus unable to dispatch another KP. This can be solved either by having dedicated runners for executing KPs that perform I/O, or by using the POSIX asynchronous I/O API.

### 4.1.1. Data structures

The scheduler's main data structures are contained in instances of the `RTE` and `actor` classes. The `RTE` class contains data related to runners and methods for KPN creation, startup, shutdown and blocking and unblocking KPs. The `actor` class is the interface between the scheduler and user-provided KP code. As such, it contains some per-KP data accessed by the scheduler and it provides the necessary glue code between the user-provided KP code and the scheduler.

#### 4.1.1.1. Runner data structures

The `RTE` class contains `static` data shared between all runners, while individual instances of the `RTE` class contain data private to each runner. The number of runners is chosen by the developer at KPN startup, usually less than or equal to the number of

CPUs. The shared data includes:

- Synchronization objects for controlling access to the shared data and key for accessing thread local storage data, where a pointer to runner's private data is stored.

- A vector of pointers to all instantiated runners, KPs, and channels (one vector for each class).

- Blocking graph, which is used for deadlock detection.

- Immutable data related to load-balancing algorithms, such as KP and channel IDs.

- Detailed accounting information.

The data private to each runner includes a run-queue, the mutex protecting the run-queue from concurrent accesses, a pointer to the currently running KP, and an instance of a random number generator. The run-queue contains KPs that are ready to run, and is organized as a doubly linked list of `actor` instances. The pointer to the currently running KP is needed so that the KP can find the data needed to return to the scheduler context when it is about to block or yield. The per-runner random number generator is needed for scalability to multiple CPUs: during early stages of development we have noticed that some implementations of the C standard library use a global state protected with a mutex, which significantly decreased performance.

The RTE methods assign unique integer identifiers to each newly created KP or channel, which are respectively created by `RTE::spawn` and `RTE::connect` methods. The identifiers start at 0 and increase by 1 for each new KP and for a channel connecting a new, previously unconnected, pair of KPs.[1] These IDs are used to map KPs and channels to vertices and edges of the load-balancing graph used by the graph-partitioning scheduling policy that we describe later.

#### 4.1.1.2. KP data structures

The `actor` class is the interface between the user-provided code defining the behavior of a KP and the scheduler. The class contains the data needed for user-mode context switch (described in the next section), and implements the trampoline code which ensures correct startup and exit of the KP. This code is reused by KPs by deriving them from the `actor` class, and by implementing their run-time behavior in the `behavior` method. When this method exits, the trampoline code ensures proper return to the scheduler, which cleans up the KP's state and ensures that the KP will not be scheduled again. Attempting to send a message to an exited KP will generate a run-time exception.

[1] In other words, multiple channels between the same pair of KPs will be assigned the same id.

### 4.1.2. User-mode context switch

Context-switch is a mechanism that enables time-multiplexing of several KPs over one OS thread (runner). Context-switch involves manipulation of low-level processor state – most notably, program counter and stack pointer – which is not accessible from high-level programming languages. Because of this, and because it is one of the most performance-critical parts of the scheduler, it is implemented in assembly language and thus inherently non-portable. Context-switch is usually performed by the OS kernel as part of scheduling processes and threads. However, applications may also use context switching in user-mode to implement "microtasks" (in our case, each KP is an individual microtask) in order to gain efficiency or predictability. These microtasks are internal to the application, i.e., invisible to the kernel, and cooperatively scheduled, which makes them easier to reason about than preemptively scheduled threads.

#### 4.1.2.1. Existing mechanisms

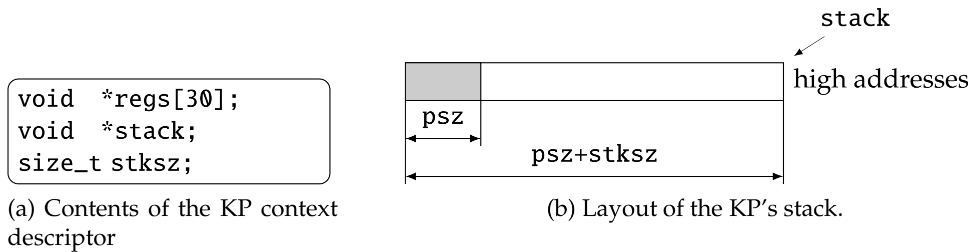
On POSIX systems, such as Linux and Solaris, the `getcontext`, `makecontext`, `setcontext` and `swapcontext` functions can be used to manipulate contexts in user-space.[2] However, they incur a large performance penalty because they save and restore the thread's signal mask, which requires an additional system call. Nevertheless, these functions are a good fall-back solution for systems where user-mode context switch is complicated to implement.

Some frameworks, e.g. YAPI [20], use `setjmp` and `longjmp` functions in a non-portable manner by assuming a particular contents of the `jmp_buf` structure, even though the C standard mandates that it must be treated as opaque. With this approach, the context switch code must also be ported to a new combination OS and CPU architecture, but this does not guarantee that the code will work. For example, newer versions of the Linux C library obscure the contents of the `jmp_buf` structure by xor-ing its fields with a thread-specific magic constant. When the fields are filled in with plain-text values, they are "decrypted" to invalid values, whose loading into CPU registers causes program crash.

The *GNU Pth* library [79] implements portable user-mode context switching by using alternate signal stacks (set up by the `sigaltstack` function) to initialize the machine context for a new thread. This library implements a full user-space thread package, but also exposes raw context switch routines, e.g., `pth_uctx_switch`. The main drawback of this approach is the high cost of setting up the context. The library will also use the context faimily of functions on platforms that support them, which results in the same drawbacks as explained above.

In section 2.1.2.1 we have described the threading building blocks library developed by Intel. Its scheduler is not suitable for our purposes because it supports only fork-join parallelism, which means that a task cannot block on events other than waiting for its child tasks to finish. Processes in a KPN, on the other hand, are not arranged in a hierarchy, and they must be able to wait on two events on channels: transition from

[2] A similar set of functions also exists on Windows: `CreateFiber`, `DeleteFiber` and `SwitchToFiber`.




(a) Contents of the KP context descriptor

(b) Layout of the KP's stack.

Figure 4.2.: KP context descriptor and stack. `psz` is the system's page size; the shaded part of the stack is inaccessible to the KP so that stack overflow is detected. We assume that the stack grows down, so the stack pointer (`stack`) is initialized to point to the highest address.

empty to non-empty state, or transition from full to not-full state. Since neither of these events coincides with KP exit, TBB's scheduler is not suitable for running KPNs written in a natural sequential style.[3]

#### 4.1.2.2. Context-switch in Nornir

Because of drawbacks of existing user-mode context switch mechanisms described in the previous section, we have implemented our own user-mode context switch mechanism optimized for the AMD64 architecture. The KP's context is stored in the context descriptor structure, shown in figure 4.2, which is a part of the `actor` class. The stack size is specified upon KP creation and cannot be changed afterwards. By default, a stack size of 12kB is used, which is enough for processes that do not use deeply recursive functions or large stack-allocated variables.

The register area stores registers necessary to resume the KP after it has been suspended. Along with stack pointer and program counter registers, we store only those registers that the calling convention mandates to be saved across function calls. This does *not* include floating-point state on the AMD64 architecture, which speeds up the context switch.

The platform-independent context manipulation routines are declared as follows:

```
void ctx_make(mctx_t *ctx, void(*start)(void*),
  void *arg, size_t stksz);
void ctx_free(mctx_t *ctx);
void ctx_swap(mctx_t *octx, mctx_t *nctx);
```

If Nornir is configured to use the optimized context switch, the low-level function `ctx_init_internal` and `ctx_swap_internal` are additionally defined. These functions are written in assembler and are the only functions specific for the AMD64 architecture.

[3]Technically, it would be possible to rewrite each KP in a continuation-passing style, where a single sequential control flow would be chopped up into multiple tasks at each (potentially) blocking call (send or receive). This difficulty is common for programs written and event-driven style and has been named *stack ripping* [80].

The ctx_make function allocates the context descriptor and stack, which can be later deallocated with the ctx_free function. ctx_make enlarges the specified stack size by one page and allocates it using the mmap system call. Then it initializes the context descriptor and stack as follows: the stack pointer is set to point to the end of the allocated stack area (because the stack grows towards lower addresses on AMD64), while the rest of the context is set up such that the first execution of the context behaves as if entry(arg) were called.

Stack overflow is an undesirable event that may silently corrupt the state of the program. In order to protect against stack overflow, ctx_make allocates an additional page at the bottom of the stack, as illustrated in figure 4.2, and uses the mprotect system call to make it inaccessible to the KP. If the KP tries to use more stack than has been allocated, it will access this page, which will in turn generate a segmentation fault and crash the program instead of silently corrupting the data.[4]

The ctx_swap function stores the context of the calling KP into the context structure pointed to by the octx argument, and restores the new context from the nctx argument. When nctx has been restored, it will appear as if the ctx_swap function has just returned.

### 4.1.3. Control flow

Nornir is started by a call to the RTE::start method, which takes the pointer to the root process as its only argument. The method reads the number of CPUs to use, scheduling policy and default channel capacity from the KRTE_POLICY environment variable, which is further explained in section 4.5. Then it proceeds to create runners, one for each CPU, and sets their affinities so that each runner is bound to a different CPU. While the affinity setting eliminates interference between Nornir's threads, they will nevertheless share their assigned CPU with other processes in the system, subject to standard Linux scheduling policy.[5]

#### 4.1.3.1. Startup

Each runner has the same startup procedure, which executes the following steps:

1. It initializes per-runner data structures (the run queue) and sets the thread-specific data pointer to point to the runner instance.

2. It tests whether it is the first runner, and if so:

    - It executes the root process, which creates the network, i.e., processes and channels.

[4] Actually, it will invoke the SIGSEGV handler; if no handler is installed the program just crashes. In our opinion, an early and visible program crash is much better than silent data corruption which *might* (or might not) crash the program, and that at some place far from where the fault happened.

[5] It is difficult to have fully "idle" system because the kernel spawns some threads for its own purposes. Using POSIX real-time priorities would eliminate most of this interference, but would not represent a typical use-case.

- If the graph-partitioning policy has been selected, it also performs the initial partitioning and migrates KPs to their assigned runners.

   This code section is protected by a single global mutex, and other runners are waiting on the associated condition variable during this time.

3. The first runner signals the condition variable and unlocks the mutex. Now all runners proceed together to enter the scheduling loop.

Using thread-specific data is the simplest way in which a KP can obtain a pointer to the runner on which it is running. This pointer is needed in situations when a KP needs to perform an action related to scheduling – yield, block itself, or unblock another KP.

#### 4.1.3.2. Scheduling loop

The scheduling loop performs the following steps (see also figure 4.3):

1. It calls the policy method to choose the next process to dispatch [CHOOSE]. If the policy method returns `NULL`, there are no more ready processes, so the runner terminates [EXIT].

2. The runner's current CPU usage is recorded [ACCT].

3. The selected process is dispatched on the CPU [KP].

4. The process has returned to the scheduler because it has blocked, yielded or exited. The runner's current CPU usage is again recorded [ACCT], and the difference between this and previously recorded CPU usage is added to the KP's CPU usage.

5. Depending on the reason for returning to the scheduler, the following is done [HANDLE]:

   - On *yield*, the process is inserted at the back of its runner's queue.[6]
   - On *block*, the process is blocking on a channel, and deadlock detection and resolution has already been done. The mutex protecting the channel is unlocked.
   - On *exit*, the KP's state (context descriptor and stack) is deallocated.

6. Go to step 1.

[6] As we shall explain later, we use mutexes to protect run-queues instead of a non-blocking queue described in [47]. The latter does not support push-to-back, so yield would not be implementable.

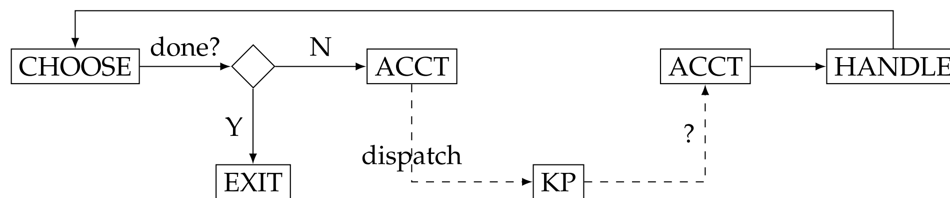



Figure 4.3.: Runner control flow. The dashed lines denote context switch to and from a KP. The "?" denotes one of the three possible reasons for the KP returning to the scheduler: yield, block or exit. The exact reason is determined by the HANDLE step.

#### 4.1.3.3. Critical sections

Critical sections ensure that a piece of code that accesses shared data is executed by at most one runner at any given time. A critical section is entered by locking the mutex associated with data, and exited when the mutex is unlocked. The critical sections and associated mutexes are as follows:

- A single mutex protecting the RTE startup.
- A single mutex protecting data structures related to graph partitioning.
- A per-runner mutex protecting its run-queue.
- A single mutex protecting the blocking graph (used for deadlock detection).
- A per-channel mutex protecting its internal buffers, blocking and unblocking of KPs at its endpoints.

The first three types of mutexes are always accessed from a runner context and will *block* the runner(s) attempting to access them when taken.

The last two types are always accessed from a KP context and use *busy-waiting*. A KP always tries to acquire the mutex without blocking, and if the mutex is already taken, the KP yields to another KP. This process repeats in a loop until the KP has succeeded in acquiring the mutex. Busy-waiting thus allows a runner to execute another KP instead of idling while waiting on the mutex to become unlocked.

Unlocking a busy-wait mutex does not affect the KP doing the unlocking. The unlocking KP just continues to run, and the KPs which are trying to acquire that mutex will run when chosen by the scheduler.

#### 4.1.3.4. Termination

All runners share a counter that reflects the total number of ready KPs in the network. The counter is set to 0 in the `RTE::start` function, it is incremented every time a KP is spawned or unblocked, and decremented every time an KP blocks or exits. The counter is incremented and decremented by using CPU's atomic primitives, which eliminates the need for a mutex. Since an exited or a blocked KP cannot unblock itself or any other

KP, this counter is used as the global termination condition: when it has reached 0, all runners will exit.

The `RTE::finish` method waits that all runners exit, cleans up the global state, and writes the collected accounting data to the stream given as the argument, default being standard output.

## 4.2. Load-balancing methods

The goal of load-balancing is to evenly distribute KPs, i.e., their CPU load, over runners in order to minimize the total running time. Load-balancing, however, introduces additional costs through three potential sources of run-time overheads:

- The overheads of tracking CPU consumption of individual KPs,
- the overheads of computing assignment of KPs to runners, and
- the overheads of migrating KPs to other runners.

We have implemented three different load-balancing methods in Nornir, each offering different trade-offs between the above overheads and the expected run-time application performance.

The first method, *static assignment*, is described in section 4.2.1. It relies on the developer's knowledge about average CPU consumption of KPs and lets the developer assign KPs to runners during KPN construction. After the KPN has started, the KPs will not be migrated to other CPUs. This is a very simplistic method, and it does not incur any of the above-mentioned run-time overheads.

The second method, *work stealing*, is described in section 4.2.2. This is a widely used load-balancing algorithm that does not need to track CPU consumption of individual KPs, and that has minimal costs of computing new assignment of KPs to runners. However, with fine-grained parallelism, it can incur significant overheads in terms of KP migration.

The third method, *graph partitioning*, is described in section 4.2.3. In addition to distributing CPU load across runners, this method also attempts to reduce communication between KPs executing on different runners and to reduce number of KP migrations. This method needs to track CPU consumption of individual KPs as well as communication volume between each pair of KPs. Its cost of computing assignment of KPs to CPUs is high, but the cost of KP migration on multi-core machines is lower than with work-stealing, especially for fine-grained parallelism. We have implemented this scheduling method to evaluate its applicability to distributed KPN run-times where communication and KP migration are orders of magnitude higher than on multi-core systems.

### 4.2.1. Static assignment

With static assignment, KPs are assigned and pinned to runners at KPN creation. Once the KPN is running, KPs are never migrated to other runners. In other words, this is

not a *dynamic* load-balancing method, and it requires that developers correctly estimate the average CPU consumption for each KP, so that they can evenly distribute the load over runners. Profiling tools like cachegrind [81] may help with such estimations.

Of the three methods, this is the only method where a runner does not use busy-waiting when it encounters an empty run-queue. Instead, it uses a semaphore to count the number of ready KPs in its run-queue. Each time a KP is inserted into the runner's queue (because it has been spawned or unblocked), the semaphore's value is incremented. Similarly, when the currently running KP blocks, the semaphore's value is decremented. When a runner is about to dispatch a new KP, it will first try to decrement the semaphore's value; if there are no ready processes, the value will be 0, and the runner will block.

Compared to busy-waiting, this scheme is friendlier towards non-dedicated environments, i.e., scenarios where Nornir runs along side of other applications. However, when runners are rapidly switching between the running and sleep states, the overhead of the kernel's sleep/wakeup mechanism considerably degrades application performance.

### 4.2.2. Work stealing

The work-stealing algorithm was first designed for scheduling fully-strict computations [82, 30], also called fork-join computations, but was subsequently extended to handle arbitrary concurrent computations [47]. We first describe the algorithm and its integration with Nornir, followed by a discussion of the algorithm's potential unfairness which may make it unsuitable for scheduling non-terminating KPNs. Then, we present a simple modification to the original algorithm which significantly benefits a common class of workloads without degrading performance of other regular workloads. These workloads and evaluation results are given in section 5.3.2. Lastly, we define fully-strict computations and present a summary of theoretical results derived by Blumofe et al. [30, 47].

#### 4.2.2.1. Algorithm description

Both work-stealing algorithms [30, 47] follow the same general idea. Each runner takes ready KPs from the *front* of its own run-queue, and also puts unblocked KPs at the *front* of its run-queue, as shown in figure 4.4. When the runner's own run-queue is empty, the runner enters a busy-waiting loop in which it chooses a random runner ("victim") and tries to steal a process from the *back* of its run-queue. The loop can finish in two ways:

1. The victim's run-queue is not empty, so the steal attempt succeeds and the stolen KP is dispatched.
2. There are no more ready KPs, so the program is finished, and the scheduler exits.

If neither is the case, the runner calls the `sched_yield` function to *yield* before starting the next loop iteration.

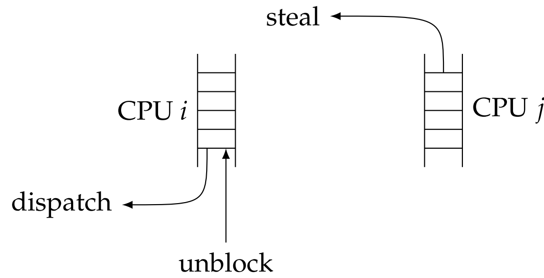



Figure 4.4.: Queue operations in work stealing. CPU $i$ accesses its own queue only at the front, while it steals KPs from other CPUs only from the back.

All queue operations have constant-time (O(1)) complexity in the *uncontented* case, i.e., their running time is independent of the number of KPs in the queue. The early algorithm [30] uses mutexes to protect the run-queues, so the complexity of queue operations can become as bad as $O(P)$, $P$ being the number of runners. This happens in the (unlikely) case where $P-1$ runners have empty run-queues and they simultaneously choose the same runner as the victim.

The later algorithm [47] uses non-blocking algorithms to implement queue operations. The worst case of $P-1$ runners accessing the same run-queue can still occur, but the contention is now handled in hardware. Depending on the machine's bus architecture, aborting memory accesses of the competing CPUs and notifying them about contention may have sublinear complexity. A potential disadvantage of this algorithm is that the non-blocking queue is implemented with fixed-size arrays, which limits the number of processes that an application can create during its execution.

#### 4.2.2.2. Integration with Nornir

As already mentioned in section 4.1.3.3, the run-queues in our scheduler are protected with mutexes, which significantly simplifies the implementation. Furthermore, supported by the work of Saha et al. [83], we deemed that this will not significantly impact performance of work-stealing on our target machine. More concretely, they have reported that even a single queue protected by a single, central lock does not hurt performance on up to 8 CPUs. This is clearly a less scalable architecture than ours, where each CPU has its own queue protected with a dedicated mutex.

The LIFO way of accessing own run queues is contrary to the FIFO policy usually used in OS schedulers, which have to be fair towards all jobs in the system. LIFO policy is fair towards all *terminating* jobs and for certain non-terminating jobs (see the next section for details). The reasons for accessing the run-queues in LIFO manner are several [84]:

- It reduces contention by having stealing threads operate on the opposite end of the queue than the thread they are stealing from.
- It works better for parallelized divide-and-conquer algorithms which typically

generate large chunks of work early. Thus, the older stolen task is, the more likely it is to provide large chunks of work to the stealing thread.

- Stealing a process also migrates its future workload because all processes spawned or unblocked by the stolen process will execute on the same CPU.
- The last unblocked or spawned process will be first to execute on that CPU, which helps to increase cache locality.

Contention is reduced only if an implementation uses a non-blocking queue implementation of [47], or two-lock queue such as that of Michael and Scott [85]. However, because we have chosen to protect each run-queue with a mutex, and because KPNs are static (i.e., new KPs are not created at run-time), Nornir does not benefit from the first three advantages. Nevertheless, accessing run-queues in LIFO manner helps with increasing locality in message-passing programs: since the arrival of a message unblocks a KP, placing it at the front of the ready queue increases probability that the message contents will remain in the CPU's caches.

#### 4.2.2.3. Fairness considerations

For jobs that *terminate*, the LIFO policy is fair: since a job terminates after all of its KPs have terminated, every KP will eventually be run.

If we allow *non-terminating*, i.e., infinitely-running, jobs, it is possible to construct a pathological case where LIFO policy is unfair and thus becomes unsuitable for executing KPNs. The simplest example consists of one runner thread and two cooperating KPs that communicate only with each other, thus preventing other KPs from running. Nevertheless, there exists a class of non-terminating jobs, called *dynamically connected systems* [58], whose communication patterns guarantee fair execution even under an unfair scheduling policy.

#### 4.2.2.4. A simple improvement

When a KP is unblocked, the original work-stealing algorithm will insert it at front of the run-queue belonging to the runner that executes the KP *doing* the unblocking. We have modified this strategy so that a newly unblocked KP will be inserted at the front of the run-queue belonging to the *last runner that executed the KP*. This strategy actively distributes the work instead of making CPUs to look for more work. With this modification, a KP can be migrated to another CPU only by being stolen, whereas in the original algorithm it can be migrated by both being stolen and by being unblocked. In section 5.3.2, we experimentally show that this modification leads to significant performance improvements on certain workloads.

This modification also invalidates all of the previously stated reasons for accessing the run-queues in LIFO manner. This raises a question, not investigated in this thesis, about whether accessing the run-queues as LIFO has any advantages for scheduling KPNs,

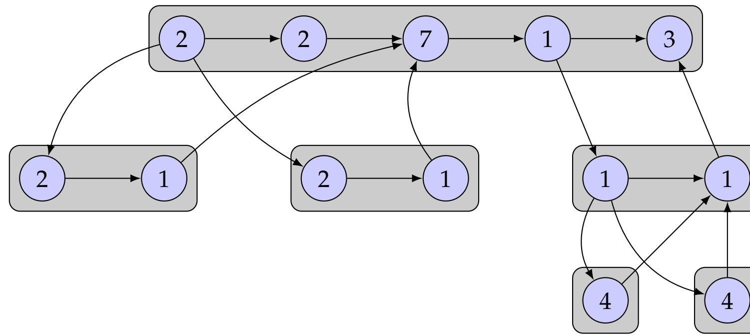



Figure 4.5.: Example of fork-join parallelism. Rectangles represent threads and circles represent tasks within a thread, together with their computational cost. Downward arrows denote forks, upward arrows denote joins, and horizontal arrows denote sequential execution of tasks within a thread. In this example, $T_1 = 31$ (total amount of work), and $T_\infty = 22$ (length of the critical path).

and whether we should use the FIFO policy instead. Doing so would have at least one advantage: unconditional guarantee of fair scheduling even for infinitely-running jobs.

We also note that this modification cannot be used if the run-queues use the non-blocking queue of Blumofe et al. [47] because its design cannot handle concurrent insertions at the front of the queue.

#### 4.2.2.5. Theoretical results

A fork-join computation, sometimes also called a fully-strict computation, is a restricted form of concurrency which is based on the `fork` and `join` primitives:

- The `fork` primitive spawns a new (child) thread.
- The `join` primitive suspends the calling thread (parent) until all of its children have exited, after which the returned results are collected.

The threads in a fork-join computation are composed of smaller sequential tasks and otherwise run to completion, without blocking at any point of their execution other than *join*. Figure 4.5 shows a fork-join thread graph, which is is typical of recursive divide-and-conquer algorithms. In a divide-and-conquer scheme, the parent divides the input and merges the results received from children, both of which are usually cheap operations, while the children perform the computationally most intensive parts of the computation.

This restricted class of concurrent programs has received much attention because it admits provably efficient schedules. It is thus employed by a number of libraries and language extensions, such as Intel's threading building blocks, Java fork-join framework [86], KAAPI [87], or Cilk [82].

Several results have been proven for the work-stealing algorithm applied to scheduling *fully-strict* computations on *P dedicated* CPUs with run-queues protected by *mutexes*.

The results are characterized with the help of the following quantities. $T_1$ is execution time on one CPU (total amount of work), and $T_\infty$ is is the length of the critical path, i.e., the time spent in the non-parallelizable part of the code (see figure 4.5 for an example). $T_\infty$ can be also interpreted as the program execution time on a hypothetical configuration with infinitely many CPUs, or as the largest sum of task execution times along any path in the task graph. In other words, $T_\infty/T_1$ is the application's average parallelism level.

- The total *expected* delay caused by processors waiting on run-queue mutexes is proportional with the total number of steal attempts $M$ made by all processors. For any $\epsilon > 0$, the delay is bounded by $O(M + P\lg P + P\lg(1/\epsilon))$ with probability $\geq 1-\epsilon$ (lemma 5 of [30]). *This is a general result, not restricted to fully-strict computations.*
- The *expected* running time of a fully-strict computation, including scheduling overheads, is $O(T_1/P + T_\infty)$. For any $\epsilon > 0$, the execution time is bounded by $O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$ with probability $\geq 1 - \epsilon$ (theorem 12 of [30]).
- The *expected* total increase in the program's space usage is $O(PT_\infty S_m)$ where $S_m$ is the size of the largest stack frame in the computation.[7]

The work-stealing algorithm was extended to use *non-blocking* run-queues, and the analytical development was extended to cover *non-dedicated* environments and *general* workloads [47]. They have proven that, when the scheduler yields between steal attempts, the *expected* execution time is $O(T_1/P_A + T_\infty P/P_A)$, where $P_A$ is the average number of CPUs on which the computation executes. For any $\epsilon > 0$, the execution time is bounded by $O(T_1/P_A + (T_\infty + \lg(1/\epsilon))P/P_A)$ with probability $\geq 1-\epsilon$. When $P_A = P$, i.e., in a *dedicated* environment, the expected execution time simplifies to $O(T_1/P + T_\infty)$, and the probabilistic execution time bound simplifies to $O(T_1/P + T_\infty + lg(1/\epsilon))$. Since the authors consider general computations described as DAGs,[8] instead of the restricted class of fully-strict computations, no space bounds could be derived.

### 4.2.3. Graph partitioning

The work-stealing algorithm does not take into account the cost of communication between processes on different CPUs, nor does it account for the cost of process migration. Load balancing methods based on graph partitioning try to find a good compromise between three conflicting goals: even distribution of load across CPUs, infrequent KP migrations, and small communication volume between KPs running on different CPUs. The last two objectives are particularly interesting in distributed systems where communication cost and KP migration between machines is orders of magnitude slower than within a single machine.

[7] This theorem is given for completeness. Fork-join computations dynamically create new processes, so it is important to show that a scheduling strategy will not cause excessive memory consumption. Since we consider only static KPN topologies, this result is of little significance to us.

[8] In practice, no program can execute infinitely. Thus, even directed graphs containing cycles can be algorithmically "unrolled" into DAGs.

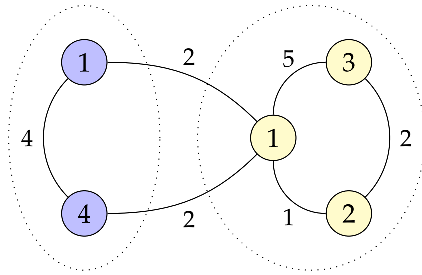



Figure 4.6.: Example communication graph. The processes have been partitioned into two sets of weights 5 and 6, and the total cost of communication between the two partitions is 4.

Load-balancing based on graph partitioning takes as input an undirected weighted graph where vertices represent concurrently running processes and edges represent communication between pairs of processes. The weights are assigned as follows:

- Vertex weight is proportional to the CPU requirements of the process.
- Edge weight is proportional to the total communication intensity (in both directions) between the two processes.

A partition of the vertex set into disjoint subsets corresponds to placing processes on different machines or CPUs, as exemplified in figure 4.6. The total weight of edges between vertices in different partitions corresponds to the total amount of communication across partitions, which is more costly than communication within a single partition.

Before reading the rest of this section, we encourage the reader to consult appendix A for a review of basic graph-theoretic notions. For completeness, we also give a survey of graph partitioning and related algorithms in appendix B.

#### 4.2.3.1. Algorithm description

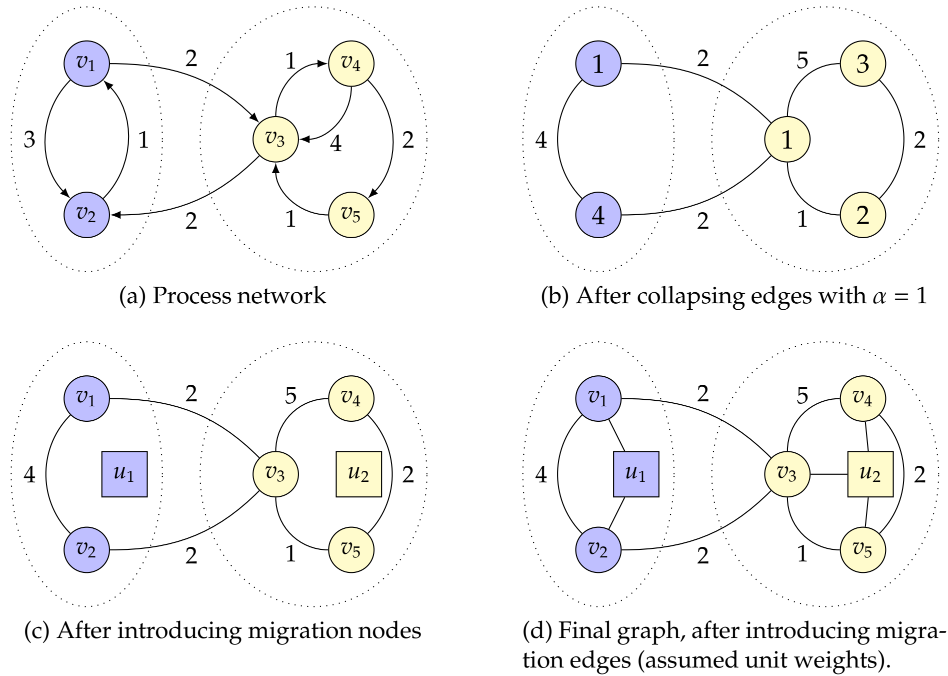
In Nornir, we follow the main idea of Devine et al. [31, 32][9] whose algorithm tries[10] to minimize the following objective function:

$$\alpha t_{comm} + t_{mig}$$

where $\alpha$ is a user-specified factor expressing a trade-off between communication and migration costs. Their algorithm cannot be directly applied to the KPN graph because it assumes a single communication path between two processes, whereas KPNs can have several. Thus, the KPN graph must be transformed into another graph that is suitable input to Devine's algorithm. This graph, which we call *rebalancing graph*, is constructed as follows:

[9] Their algorithm actually operates on *hypergraphs*, described in appendix A. Since graphs are sufficient to model KPNs, we have simplified our description by restating their algorithm in terms of graphs.

[10] Finding a partition that actually *does* minimize the objective function is NP-hard and thus infeasible to solve in realistic time, except for very small graphs. See appendix B for details.




(a) Process network

(b) After collapsing edges with $\alpha = 1$

(c) After introducing migration nodes

(d) Final graph, after introducing migration edges (assumed unit weights).

Figure 4.7.: An example of transforming a KPN into a rebalancing graph with $\alpha = 1$.

1. The KP and channel accounting data is used to set weights on the original KPN graph (figure 4.7a).

2. Edges between the same pair of nodes are collapsed into a single edge whose weight is the sum of weights of constituent channels multiplied by $\alpha$ (figure 4.7b).

3. $n$ new nodes $u_1 \dots u_n$, one for each CPU, are introduced (drawn as squares in figure 4.7c). These nodes are *fixed* to their respective partitions in the partitioning process.

4. Fourth, if a node $v_i$ is associated with a task currently assigned to CPU $k$, a *migration edge* is created between $v_i$ and $u_k$; its weight is set to the cost of migrating data associated with process $v_i$ (see figure 4.7d).

Applying a graph partitioning algorithm to the rebalancing graph yields the new assignment of KPs to runners.

Devine's algorithm and our adaptiation take an *existing* assignment of KPs to runners and produce a new assignment, which poses the problem of creating the *initial* assigment. One possible approach is to let developers manually assign KPs to runners. We have chosen an approach that relies on detection of load imbalance potentially created during the initial assignment. Once detected, the rebalancing routine will be invoked to equalize the load among CPUs.

Since the initial assignment is performed at KPN startup, the CPU usage of KPs and communication intensities over channels are unknown. We thus assume that all KPs use the same amount of CPU time and that communication intensity over all channels is the same. We then transform the KPN graph into an undirected graph as described above, but stopping after collapsing edges (step 2). We then assign unit weights to vertices and edges and invoke a graph partitioning algorithm to obtain the initial assignment.

#### 4.2.3.2. Integration with Nornir

Devine et al. assume that the application itself invokes the load-balancing algorithm when its work distribution has changed significantly. This approach works for certain classes of applications, such adaptive simulations, that run in well-defined iterations. These applications will invoke the load-balancing algorithm after every $k$'th iteration, where $k$ is specified by the developer. On the other hand, there are many applications, e.g., case-studies presented in section 3.3, whose CPU usage patterns vary continuously. Such applications would have to implement self-monitoring to decide when the load-balancing algorithm should be invoked.

In order to free developers from providing each application with self-monitoring, we have implemented a heuristic that attempts to detect load imbalance. The heuristic is integrated with the scheduler's main loop: it executes during the scheduler's [CHOOSE] step (see figure 4.3), where it monitors the idle time $\tau$ *collectively* accumulated by all runners. When a runner attempts to dequeue a process while its run-queue is empty, it updates the collective idle time and continues to check the run-queue, yielding (`sched_yield`) between attempts. Whenever *any* runner succeeds in dequeuing a KP, it resets $\tau$ to 0. However, if $\tau$ exceeds a threshold set by the developer, the load-balancing algorithm is invoked. After the algorithm has finished, KP and channel accounting data are reset to 0, in preparation for the next load-balancing.

After repartitioning, we avoid bulk migration of KPs because it would require locking of all run-queues, migrating KPs to their new runners, and unlocking run-queues. The complexity of this task would be linear in the number of ready KPs in the system, so runners could be delayed for a relatively long time in dispatching ready KPs, thus decreasing the total throughput.

Instead, KPs are only reassigned to their new runners by setting a field in their control block (a part of the `actor` class), but without physically migrating them. Each runner takes ready processes *only* from its own queue, and if the KP's run-queue ID (set by the rebalancing algorithm) matches that of the runner's, the KP is run. Otherwise, the KP is reinserted into the run-queue to which it has been assigned by the load-balancing algorithm.

In our implementation, we have used the state-of-the-art PaToH library [88] for graph and hypergraph partitioning, which is distributed in binary-only form. During our experiments, we have experienced some issues with this library. Nornir measures CPU consumption in nanoseconds, which quickly generates large CPU usage numbers. PaToH, however, uses internally only 32-bit arithmetic. Therefore, it cannot handle large vertex and edge weights, so it exits with an error message about detected integer

overflow. We have first tried to solve this problem by dividing all vertex weights by the smallest vertex weight. This approach did not work because of the large dynamic range of vertex weights, so it would still happen that the resulting weights are too large. We considered and tested two approaches of handling this situation:

- Run the partitioning algorithm more often, or
- scale down all vertex weights.

The first approach created large scheduling overheads and made it impossible to experiment with infrequent repartitionings. We have therefore also implemented the other option and used it for all experiments of chapter 5. Here, we have transformed vertex weights using the formula $w' = \frac{w}{1024} + 1$ before handing them over to the graph partitioner. This loss of precision, however, causes an *a priori* imbalance on input to the partitioner, so the generated partitions have worse balance than would be achievable if PaToH internally worked with 64-bit integers. This rounding error may have contributed to the limited performance of scheduling based on graph-partitioning (see chapter 5 for methodology and results), but we cannot easily determine to what degree.

## 4.3. Message passing

In the Kahn model, message-passing is the *only* means of communication between KPs. As such, it must not only fulfill the Kahn semantics, but it should also be efficient. The Kahn semantics requires that receiving a message from an empty channel shall block the receiving process until a message arrives. Message sending *should* always succeed without blocking, but such definition of the send operation would require infinite channel capacities, which any real implementation obviously cannot provide. We thus limit channel capacities and define the send operation to *block* when it tries to send a message to a full channel; see section 3.1.5 for a full discussion of this issue. The send and receive operations must be coordinated: when a KP receives a message from a full channel, it must unblock the KP on the sending side (if it was blocked). Similarly, when a KP sends a message to an empty channel, it must unblock the KP on the receiving side (if it was blocked).

The message passing facility consists of the following classes: `send_port`, `recv_port` and `channel`. All three classes are templates parametrized by the type of the message that will be transfered over the channel (e.g., `send_port<int>`). This ensures compile-time safety through strong typing – if the developer tries to send or receive a message of a different type than the port has been instantiated with, the compiler will report an error.[11] These classes are respectively derived from the `port_base` and `channel_base` classes. The base classes hold common bookkeeping data, independent of what type of message will be transfered, which makes it easy to write code that treats all channels uniformly when necessary, e.g., deadlock detection.

[11] Sending dynamically-typed messages over channels can nevertheless be achieved by using libraries such as Boost.Variant [44].

KPs do not use the `channel` class directly. They define instead instances of `send_port` and `recv_port` classes, which ensure unidirectional communication. The ports are connected together by the root process with the help of `RTE::connect` method, which creates an instance of the `channel` class with the given initial capacity. The channel class uses a circular buffer to store messages, and it defines only *non-blocking* send and receive methods. Such abstraction allows us to easily change the transport mechanism without having to modify the rest of the code, which allows easy experimentation with different transport mechanisms, such as message queues or sockets.

Messages are sent by invoking `send` and `recv` methods on ports; these methods use the channel's non-blocking methods to implement the blocking semantics. The `send` method accepts an optional data volume parameter, so the KP can provide accurate information about the data volume sent over the channel to the graph partitioning scheduling policy. The parameter's default value is 1, which just counts the number of messages transfered over the channel. The `send` and `receive` methods use `RTE::wait` to block the calling KP and `RTE::signal` method to unblock another KP.

We have also extended channels with support for *EOF indication*:[12] the sender can call the `set_eof` method to set the EOF status on a channel when it has no more messages to send. After this, the receiver will be able to read the remaining buffered messages. When these messages have been exhausted, the first call to `recv` method will return `false` without blocking and without changing the target message buffer; the second call will block the receiver forever. Introducing EOF signaling has the significant advantage of not having to designate a specific value of the type as EOF indication. Indeed, all values of a type (e.g., `int`) might be meaningful in a certain context, so no value is available for encoding the EOF message. In such cases, one would be forced to use solutions that are more cumbersome to use and impose additional overhead, such as dynamic memory allocation with NULL pointer value representing EOF.

As mentioned in section 4.1.3.3, the channel's buffer is protected from concurrent accesses by a busy-wait mutex. We have also considered to use spin-locks for protecting the channels, but we have rejected this because the time to send or receive a message is proportional with message size.[13] Thus, if copying would take a long time, yielding gives another KP a chance to do some useful work.

## 4.4. Deadlock detection

Channels in Nornir have finite capacites and `send` operation blocks on a full channel. This modification of the Kahn semantics may lead to artificial deadlock, i.e., cause the KPN to terminate earlier than with infinitely-sized channels; see section 3.1.5 for a full discussion. To overcome the problem of artificial deadlock, we have implemented a

[12]EOF stands for end-of-file. We deemed the name appropriate since channels do resemble files to a certain degree.

[13]Technically, spinlocks also use busy-waiting, but they are provided by the system and not integrated with our scheduler. Spinlocks would thus have the same effect as blocking mutexes: a runner would waste time in waiting for the spinlock to become unlocked, and would be unable to run another KP.

simple centralized deadlock-detection and resolution algorithm based on a blocking graph, also called the wait-for graph [5]. In the next two sections we describe the blocking graph, prove a property of it that greatly simplifies the implementation, and discuss integration of deadlock detection with Nornir.

### 4.4.1. Blocking graph

The deadlock detection and resolution algorithm is based on finding cycles in the blocking graph. Vertices of the blocking graph correspond to KPs, while edges describe the *wait-for relationship* between KPs. Whenever a KP $p_1$ blocks on a channel with KP $p_2$ on the other end, a directed edge $(p_1, p_2)$ is added to the blocking graph.

The restrictions on communication patterns imposed by the Kahn model allows us to prove the following fact about the shape of the blocking graph: every vertex in the blocking graph has at most one outgoing edge, and the existence of a cycle in the blocking graph is a sufficient condition for deadlock to exist. To prove this, we restate the pertinent restrictions of the Kahn model:

1. A KP can block on at most one channel.
2. Channels are 1:1, i.e., there is exactly one KP at each end.

From these restrictions and the definition of the blocking graph, it follows that each vertex can have at most one outgoing edge. The second restriction additionally implies that a KP can be unblocked *only* by the KP it is waiting on. In other words, if a deadlock occurs, it cannot be broken by KPs outside of the deadlocked cycle because they cannot access the channels involved in the deadlock.[14] Thus, all paths in the blocking graph have the topology of either a simple line, or when a deadlock occurs, a ring. This property greatly simplifies the implementation of the deadlock detection algorithm, described in the next section.

### 4.4.2. Integration with Nornir

In the current implementation, the blocking graph is represented with a hash table that uses `actor` pointers for keys and values. The data structure and the algorithm are centralized and protected by a single mutex, as explained in section 4.1.3.3.

Deadlock detection is invoked from the `RTE::wait` method. Whenever KP $p_1$ would block on KP $p_2$, either on send or receive, the algorithm starts following the path in blocking graph starting from $p_2$. As the algorithm iteratively follows the edges of the blocking graph, it also remembers the smallest full channel along the path. This process finishes with one of the following outcomes:

- The search ends at a KP $p'$ without an outgoing edge in the blocking graph.
- The search ends at a KP $p'$ which is waiting on $p_1$, i.e., the edge $(p', p_1)$ exists in the blocking graph.

[14] In the more general communication model (m:1, 1:m, or m:n), this would not be the case.

In the first case, no deadlock exists. The edge $(p_1, p_2)$ is *added* to the blocking graph and $p_1$ is blocked. In the second case, a deadlock would occur if $p_1$ were allowed to block. Instead of just blocking $p_1$, the algorithm first increases by one the capacity of the smallest full channel in the cycle, as suggested by [73]. If this channel is the same that $p_1$ tried to send data to,[15] $p_1$ is resumed. Otherwise, $p_1$ is blocked, the KP at the send side of the enlarged channel is unblocked, and the corresponding edge removed from the blocking graph.

Similarly, whenever the `RTE::signal` method unblocks a KP $p$, the entry with key $p$ is *removed* from the blocking graph.

## 4.5. Configurability and portability

Nornir's implementation should be fully portable to POSIX operating systems, and it is also partly portable to Windows OS. Load-balancing based on graph partitioning (see section 4.2.3) does not work on Windows because the graph-partitioning library we used, PaToH [88], is distributed only in binary form, but not for Windows. This section describes the current configuration options for Nornir, and also discusses other possible, but unimplemented, options.

### 4.5.1. Scheduling options

Nornir can be configured at compile-time to use either 1:1 or m:n scheduling. The 1:1 scheduler uses one kernel thread per KP, so all KPs are scheduled by the kernel's default preemptive scheduler. The m:n implementation uses user-space context switching to time-multiplex $m$ KPs over $n$ kernel threads, where $n$ is usually set to be equal to the number of CPUs in the system. The m:n implementation is cooperative and supports several scheduling algorithms, described in section 4.2. The scheduling algorithm is selectable at run-time, before KPN startup.

When using the m:n scheduler, the context switch mechanism can be configured to use our hand-optimized context switch written in assembler for the AMD64 architecture or the `swapcontext()` family of functions (on POSIX). The latter is less efficient, but is portable to different operating systems and processor architectures. Since Windows has a more complex process/thread model than POSIX, we use the fibers API[16] instead of our own context switch, just to be on the safe side.

### 4.5.2. Accounting options

Accounting, i.e., collection of detailed performance statistics data, is a compile-time option because it causes much extra overhead. The measured cost of [send → context switch → receive] transaction on the ring benchmark drops from $\sim 1.4\mu$s to $\sim 0.8\mu$s

[15] It is perhaps not obvious that in this case the process must have attempted to *send* to the channel. This is so because the search for deadlocks considers only *full* channels, and a process cannot block by attempting to *read* from a *full* channel.

[16] This is a set of functions analogous to POSIX context functions

when accounting is disabled (see section 5.2.3.2 for details). The largest overhead in the 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.

Accounting is available only in combination with the m:n scheduler. Since the 1:1 scheduler gives us no control over how and when individual KPs will be scheduled,[17] much of the collected information is either not available (e.g., idle time, KP migrations) or it cannot be put to good use (e.g., local and remote traffic). When enabled, the accounting mechanism collects the following information:

- *Real* (wall-clock) and *CPU* (consumed CPU cycles) time used by Nornir, as well as CPU time used *only* by KPs. The timer resolution is a few nanoseconds on Solaris and Linux platforms.
- *Idle* time used by the Nornir. The idle time is the total real time that runners used spinning in a loop, trying to acquire work.
- Number of *context switches*; each dispatch of a KP counts one.
- Number of *stolen* (for work-stealing policy) or *migrated* (for graph-partitioning policy) KPs.
- Number of times a runner has encountered an *empty queue*, and the number of *steal attempts*; both values are accumulated over all runners.
- Number of *yields*, which are caused by contention over busy-wait mutexes. As explained in section 4.1.3.3, there are two sources of such contention: contention over the mutex protecting a channel, and contention over the global deadlock-detection mutex.
- *Local and remote traffic*, i.e., number of messages that have been sent to a process on the same CPU or on another CPU, respectively.
- Number of deadlock detections that have been *started*, and the number of actual deadlocks that have been *resolved*.
- Number of times the *repartitioning* routine was invoked; this is applicable only to the graph-partitioning policy.

All accounting variables are accessed using CPU's atomic instructions [5] which ensure correct concurrent updates of shared variables without using mutexes. In addition to the common test-and-set and compare-and-exchange [67] atomic instructions, the AMD64 architecture can also perform various logical and arithmetic operations atomically. Thanks to these instructions, updating accounting data is not only free of mutexes, but also efficient.[18]

[17]Recall that each KP corresponds to one OS-level thread in the 1:1 model, so it is also preemptively scheduled.

[18]The compare-and-exchange instruction takes three operands: a memory address `M`, an old value `O`, and a

### 4.5.3. Run-time options

Certain aspects of Nornir's operation are configurable at run-time for practical reasons: number of CPUs, scheduling algorithm, default channel capacity and whether deadlock detection is enabled or disabled. The default channel capacity is used for all calls to the `RTE::connect` method that do not explicitly specify capacity.

All of the above parameters are specified through `KRTE_POLICY` environment variable. The following examples specifying two different run-time configurations:

```
KRTE_POLICY=4;WS,0;64
KRTE_POLICY=4;HP,128;-64
```

The first line specifies 4 CPUs, work-stealing policy (`WS,0`), default channel capacity of 64 messages, and deadlock detection is *enabled*. The second line specifies 4 runners, graph-partitioning policy with idle time parameter $\tau = 128$ (`HP,128`; see section 4.2.3.2 for details), default channel capacity of 64 messages, and deadlock detection *disabled* (note the *negative* default capacity). The repartitioning interval is automatically multiplied by the number of CPUs to compensate for the fact that under load-imbalance conditions the idle time accumulates more quickly with more CPUs.

### 4.5.4. Abandoned options

We have also experimented with preemptive scheduling implemented purely in user-space as well as with zero-copy message passing. We have eventually abandoned both ideas for reasons described below.

#### 4.5.4.1. Preemptive scheduling

In the earliest stages of our work (not published), we have experimented with preemptive, proportional-share scheduling. We have used POSIX timers and real-time signals to implement preemption fully in user space. In our implementation, context switching was done from the signal handler, which was made possible by setting up the signal handler with the `sigaction` system-call with the `SA_SIGINFO` flag set. The kernel then delivers the interrupted context as one of the signal handler's arguments, and this context can under some OS-es, e.g., Solaris,[19] be used by the context functions described in section 4.1.2.1.

However, such approach has a significant limitation: only a single KP has access to the full POSIX API, while all other KPs are limited only to async-signal safe functions. This limitation turned out to be too serious for many real-world applications, so we have decided to support only cooperative scheduling mechanism instead. Cooperative

new value `N`. It performs the following operation *atomically* (pseudo-code): `{ if(*M==O) { *M = N; return true; } return false; }` Arithmetic operations using this instruction can be implemented atomically, but looping is needed because the instruction fails in case of contention. AMD64 arithmetic operations never fail when performed atomically.

[19]POSIX does not require that this works and a different approach is needed on Linux.

scheduling does not conflict with our requirements, and it also significantly simplifies the implementation.

#### 4.5.4.2. Zero-copy message passing

Given that Nornir runs KPs in a shared address space, it is still possible that they modify each other's state and thus ruin the KPN semantics. To minimize the possibility of this happening, there are two possibilities of implementing message passing:

- 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 send and receive operation, but is essentially zero-copy. We have chosen to use the second approach in Nornir because measurements on Solaris have shown that memory (de)allocation, despite having being optimized by using Solaris's *umem* allocator, has larger overhead than copying when messages are smaller than ~ 256 bytes.

## 4.6. Summary

In this chapter, we have presented implementation details of Nornir's components: KP scheduler, load-balancing algorithms, message passing, deadlock detection and accounting mechanisms. Nornir should be portable to POSIX operating systems and partly also to Windows. On Windows, the load-balancing algorithm based on graph partitioning is not supported because the PaToH graph partitioner is distributed only in binary form, but not for Windows. This limitation can be remedied by using a different graph partitioning library.

The KP scheduler supports two scheduling models: the preemptive 1:1 model and the cooperative m:n model. In the 1:1 model, there is a 1:1 correspondence between KPs and OS-level threads, which are in turn scheduled by the underlying OS scheduler. In the m:n model, many KPs are mapped over a single OS-level thread (runner) which maintains an own run-queue of ready KPs. In the m:n model, context switching between KPs is performed entirely in user-space. The user-mode context switch can use either the OS-provided mechanisms (available both for POSIX and Windows), or our optimized code (available only for AMD64 architecture on POSIX).

We have implemented three load-balancing methods: static assignment, work-stealing and a method based on graph partitioning. The static assignment lets the developer specify the runner on which each KP will execute. Once the KPN has started, KPs will never be migrated to other runners. Work-stealing is a widely-used load-balancing method, and we have also proposed an improvement that, as we will experimentally show in the next chapter, leads to significant performance improvements on certain

workloads. The last load-balancing method is based on graph partitioning, which tries to reconcile three conflicting goals: good distribution of load across CPUs, low message traffic between KPs executing on different runners, and infrequent KP migrations.

To enable execution in limited memory space, the message-passing mechanism imposes finite channel capacities and redefines the send operation to block on full channels. This modification of Kahn semantics opens for the possibility of artificial deadlock, a problem which we address by a centralized run-time deadlock detection and resolution algorithm. This algorithm is based on maintaining a blocking graph (also called wait-for graph) and using it to searching for cycles of blocked KPs.

Finally, we have equipped Nornir with extensive run-time accounting of various performance data. This mechanism has high overheads, so it can be completely disabled at compile-time. Furthermore, it is available only with the m:n scheduler because much of the collected data is either not available or it cannot be put to good use with the 1:1 scheduler.

In the previous and this chapter we have discussed how KPNs can be used to model parallel applications and how a KPN run-time can be implemented in practice. In the next chapter, we will extensively evaluate performance of applications when expressed as KPNs as well as performance of Nornir itself.

# 5. Performance evaluations

In the previous chapter, we have presented implementation details of Nornir. In this chapter, we will use Nornir for performance evaluations in three areas: applications, microbenchmarks, and scheduling algorithms. To evaluate application performance, we have used the word frequency and k-means problems introduced in section 3.3. For microbenchmarking and evaluation of scheduling algorithms, we have designed and implemented a set of synthetic workloads.

In section 5.1, we investigate the performance of the two case-study applications, word frequency and k-means, introduced in section 3.3. We have implemented KPN-MR and KPN-FLEX solutions for both applications (four programs in total) and compared their relative performance. We have also compared performance of the four solutions with their solutions in Phoenix [26].

In section 5.2, we describe our synthetic workloads, which we then use to evaluate performance of KPN implementation options and scalability of Nornir. With the latter evaluation, we want to find out how well Nornir scales along two dimensions: KPN size (number of KPs), and parallelism granularity. We also investigate the effects of parallelism granularity on deadlock detection and under which conditions, if at all, centralized deadlock detection limits scalability.

Finally, in section 5.3, we use the same synthetic workloads to evaluate load-balancing algorithms. We first evaluate performance of work-stealing and graph partitioning load-balancing algorithms, where we focus on two main metrics: speedup and proportion of local traffic in the total traffic. In addition, we investigate the running time variance with the graph-partitioning algorithm. Then, we focus on *quantifying* the scalability of the work-stealing method in terms of work amount per message, both with and without our modification of section 4.2.2.4.

## 5.1. Application benchmarks

To evaluate whether the KPN models have any performance advantages over MapReduce models, we have implemented the KPN-FLEX and KPN-MR variants of our case-study applications described in section 3.3. We first evaluate performance of KPN-FLEX and KPN-MR solutions for each of the two applications individually. Then, we compare performance of all four KPN solutions against their Phoenix counterparts.

### 5.1.1. Methodology

We have configured Nornir to use the m:n work-stealing scheduler with deadlock detection and detailed accounting *enabled*, and default channel capacity of 64 messages.

The benchmarks have been compiled for 64-bit mode with GCC 4.3.2 and maximum optimizations turned on (`-m64 -O3 -march=opteron`) and run on a 64-bit-bit AMD Opteron on 2.6 GHz with 4 dual-core CPUs and 64 GB of RAM running Linux kernel 2.6.27.3. Our discussion of results is based on the average of 10 measurements.

In our evaluations, we compare the *unoptimized* (in terms of the number of exchanged messages) KPN-FLEX implementation with the *optimized* KPN-MR implementation. We have decided against optimizing the KPN-FLEX implementations because optimizations would obscure similarity with the sequential algorithm and analogy with files that we explained in section 3.3.3.2.

### 5.1.2. Word frequency

In this section, we evaluate performance of KPN-MR and KPN-FLEX solutions of the word frequency application. Both solutions have been extensively described in section 3.3.2.

The non-optimized KPN-MR implementation of the word frequency program was slower by an order of magnitude than the optimized implementation for which we present results here. In the non-optimized implementation, 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. In the optimized implementation, each worker (both in Map and Reduce stages) dynamically allocates as many vectors as it has output channels and uses them to store the output key-value pairs instead of sending them to channels. Then, when the worker has processed the whole input, it sends to each output a *single* message containing a pointer to the vector with the messages destined for that channel.

We have run the word frequency program on the same dataset that is distributed with Phoenix and used for Phoenix benchmarks; it consists of three files of sizes 10, 50 and 100MB respectively. Both implementations have been run on 1, 2, 4 and 8 CPUs with the number of runners in each stage varying from 8 to 128 in steps of 8. Figure 5.1 shows running times of the KPN-FLEX and KPN-MR solutions for the three datasets. The number of workers for the KPN-MR version is *per stage*, so there are four times as many processes in total because there are 4 stages, as shown in figure 3.12a.

From figure 5.1 and table 5.1, 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. With respect to the number of workers on 2 or more CPUs, the situation is more complicated: for both implementations, the running time *decreases* as the number of workers *increases* up to a certain point. Word frequency is both CPU- and communication-intensive task (especially in the KPN-FLEX version) 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 4.3, processes acquire mutexes by busy-waiting and yielding between successive attempts. When there are enough ready processes,

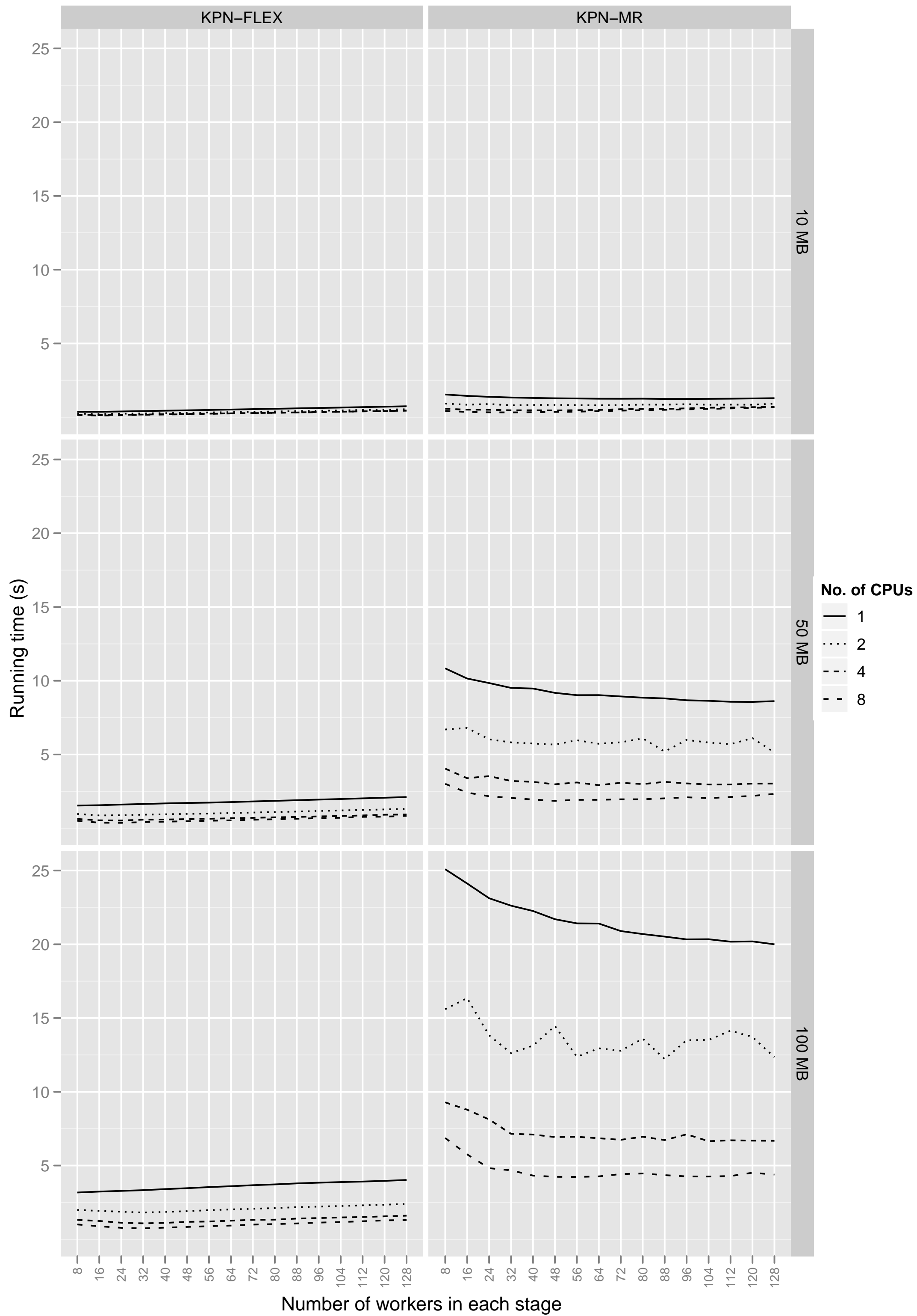



Figure 5.1.: Running times of the word frequency program on 10, 50 and 100 MB datasets, using KPN-FLEX and KPN-MR solutions.

| Size | KPN-MR | | | KPN-FLEX | | | |
|---|---|---|---|---|---|---|---|
| | $n$ | $t$ | $\alpha$ | $n$ | $t$ | $\alpha$ | $f$ |
| 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 5.1.: Word frequency experiment summary: number of workers $n$ that achieves the best running time $t$ (in seconds), relative speedup over one CPU ($\alpha$) and speedup factor of KPN-FLEX over KPN-MR ($f$).

encountering a locked mutex will yield to another 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. First, the number of context switches increases, which also indirectly degrades performance by increasing pressure on CPU caches. Second, we believe that an even more important factor is contention over the single deadlock detection mutex.

Table 5.1 shows a summary of results for each experiment configuration: the best mean running time, the number of workers for which it has been achieved, and relative speedup with respect to the case of one runner. The table reveals that 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 sublinear because of two factors: algorithmic complexity of sorting, which is $O(n \lg n)$, and the effects of deadlock detection. We see also 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*.

### 5.1.3. k-means

In this section, we evaluate performance of KPN-MR and KPN-FLEX solutions of the k-means application. Both solutions have been extensively described in section 3.3.3.

We have optimized the KPN-MR variant of the k-means application in two ways: by reducing the number of sent messages in the same way as for the word-frequency program, and by introducing an additional process to iterate the MapReduce computation, as explained in sections 3.3.3 and 3.3.3.1.

| | Points | Groups | Iterations |
|---|---|---|---|
| A | 100000 | 100 | 97 |
| B | 200000 | 50 | 140 |
| C | 200000 | 100 | 139 |

Table 5.2.: 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.

The k-means program generates a number of 3-dimensional points with random integer coordinates, all of which fall into the cube $[0, 1000)^3$. The random number generator is always initialized with the same seed, so each experiment run executes the same number of iterations. We have measured performance of the program on the three problem sizes shown in table 5.2, 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 in each stage 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 frequency example. The main reason for smaller gain in speedup is that the KPN-MR solution of k-means does very little unnecessary work, namely only sorting before computing the new means for the next iteration. Nevertheless, the overhead of sorting accumulates over all iterations of the algorithm. The speedup of the KPN-FLEX solution over KPN-MR is shown in Table 5.3. From the table, we see that the KPN-FLEX speedup factor over KPN-MR is $\sim 1.2$ for the problem sizes A and C, and $\sim 1.4$ for problem size B. 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 frequency example, the k-means program shows *almost perfect linear scalability* with the number of CPUs.

### 5.1.4. Comparison with Phoenix

We have also compared the performance of our KPN-FLEX and KPN-MR implementations with Phoenix [26], which is a MapReduce implementation designed specifically for multi-core machines. We have run Phoenix with its default settings which means that it will use as many threads in each of the Map and Reduce stages as there are CPUs on the machine. We have tried to run Phoenix with more than 8 workers in the Map and Reduce stages and found that it decreased performance.

We had to compile Phoenix programs in 32-bit mode because the code contains non-portable casts between pointers and integers. 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 used by our implementation. As a consequence, Phoenix experiences fewer data cache and TLB misses in 32-bit mode than it would in 64-bit mode.

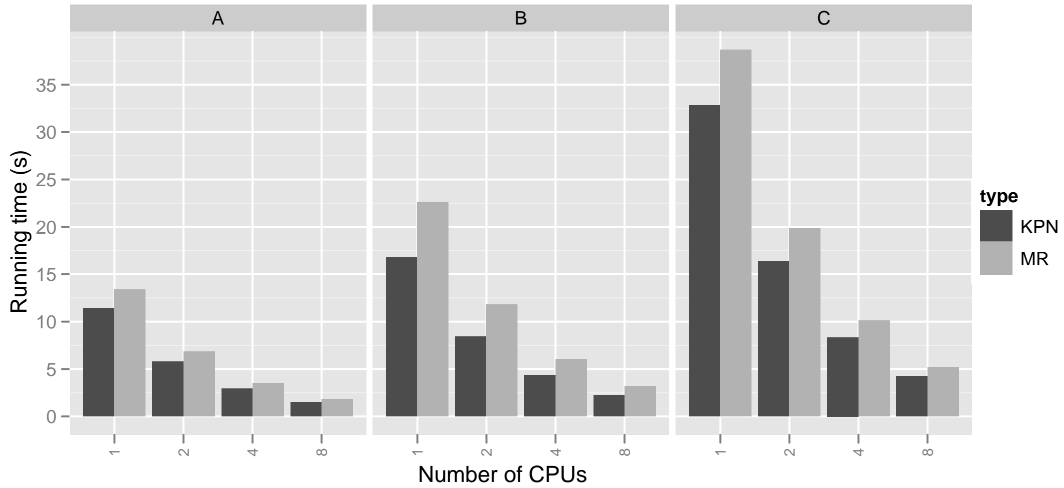



Figure 5.2.: Benchmark results for the k-means program run on three different problem sizes, implemented as KPN-MR and KPN-FLEX. KPN-FLEX solution has as many worker process as the number of runners that were used in the benchmark. Similarly, the KPN-MR solution has the same number of workers as CPUs in each stage.

| Size | KPN-MR | | KPN-FLEX | | |
|---|---|---|---|---|---|
| | $t$ | $\alpha$ | $t$ | $\alpha$ | $f$ |
| 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 5.3.: k-means experiment summary: mean running time ($t$), relative speedup over one CPU ($\alpha$) and and speedup factor of KPN-FLEX over KPN-MR ($f$).

| Size | Phoenix | KPN-MR | | KPN-FLEX | |
|---|---|---|---|---|---|
| | $t$ ($\sigma$) | $t$ | $f$ | $t$ | $f$ |
| Word Frequency | | | | | |
| 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 5.4.: Mean running times in seconds ($t$) of the word frequency and k-means programs for Phoenix, KPN-MR and KPN-FLEX; in case of KPNs, the smallest time on 8 CPUs is shown. $\sigma$ is standard deviaton and $f$ is speedup factor over Phoenix.

We have run Phoenix implementations of the word frequency and k-means programs 10 times on each problem size, and calculated the mean and standard deviation of the running time. Table 5.4 compares these results with running times of KPN-MR and KPN-FLEX solutions for the two applications. In the word frequency 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 $\sim 2$ 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 that will be run. 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 $\sim 2.7$ 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, again, performs better because it avoids doing unnecessary work. We believe that the main reason for the unexpectedly good performance of the KPN-MR solution is the avoidance of framework (de)initialization that Phoenix has to perform on each iteration. Actually verifying this conjecture would entail implementing extensive changes to Phoenix in order to make it support iterative MapReduce computations without framework (de)initialization on each iteration.[1]

### 5.1.5. Conclusions

In this section, we have implemented the word frequency and k-means applications in two variants. The KPN-FLEX variant implements a KPN suited to the problem, while the KPN-MR variant implements a KPN that closely follows the MapReduce semantics.

[1]Implementing the opposite option – making KPN-MR solution go through framework (de) initialization on each iteration – would answer the (pointless) question "How slow can KPN-MR run?". Our conjecture, however, actually asks the opposite question: "How fast could Phoenix run?"

Our experiments have shown that KPN-FLEX outperform the KPN-MR variants by a factor of 2.95 – 6.74 on the word frequency application, and by a factor of 1.17 – 1.43 on the k-means application. Furthermore, the KPN-FLEX solutions outperform also their Phoenix counterparts by factors of 2.65 – 2.76 and 1.28 – 1.75, respectively.

We can thus conclude that the flexibility of the KPN model gives developers an opportunity to develop applications that perform better than when implemented in the MapReduce model. The main reason for this is that the MapReduce model is rather rigid, because it requires that the problem be cast into the key-value paradigm, which often leads to unnecessary work at run-time (most notably, sorting). In the KPN model, data representations, algorithms and workflow (reflected in the KPN graph) can be freely matched to the problem at hand.

We have also made some conjectures about finer details of performance characteristics that have emerged from our measurements: sublinear speedup of the word frequency application as new CPUs are added, and the unexpected result of the KPN-MR solution of k-means outperforming the Phoenix solution. We have not followed them up because our main goal here was to evaluate two *models* of parallel programming – the performed experiments have been sufficient to show that the more flexible KPN model, unlike the MapReduce model, indeed does give many optimization opportunities.

## 5.2. Microbenchmarks

In this section, we evaluate performance of KPN implementation options and scalability of Nornir. For this, we have designed and implemented a set of synthetic workloads that we have used to perform two sets of experiments.

In the first experiment set, we have used these workloads to investigate the relative performance of implementation options for scheduling and message-passing. We focus on overheads of scheduling, scalability with the number of KPs, and performance of message-passing. Since we have performed these experiments in the early development stages, we ran them on a small dual-core machine.

For the second experiment set, we have configured Nornir to use our m:n scheduler with the work-stealing scheduling algorithm and our optimized message transport mechanism. With these experiments, we have evaluated scalability or Nornir along two dimensions: parallelism granularity and effects of the centralized deadlock detection algorithm on performance. We have compiled and run these experiments on the same hardware and OS as described in section 5.1.1.

This section is organized as follows. We first describe our synthetic workloads, then we proceed to present the results of each experiment set separately, and we conclude with a summary of results and possible improvements to Nornir. Presentation in each of the two experiment sections is organized similarly: we first describe the experimental methodology, then we present results for each of the tested workloads, and conclude with a discussion of the results.

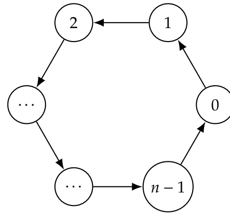



Figure 5.3.: KPN for the "ring" benchmark.

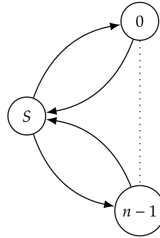



Figure 5.4.: KPN for the "AES" benchmark.

### 5.2.1. Workloads

We have designed workloads to measure different aspects of Nornir, such as scheduling and message-passing overheads and scalability with the KPN size. In most benchmarks, KPs just consume CPU cycles. The exceptions are AES, which is both CPU- and memory-intensive, and ring, where KPs do not perform any kind of processing over messages.

#### 5.2.1.1. Ring

The ring KPN (see figure 5.3) is designed to measure message passing and scheduling overheads, and is modeled after the *lat_ctx* benchmark from the *lmbench* suite [89].[2] $n$ KPs, $0 \ldots n-1$, are connected into a ring topology, where each KP receives a message on its input channel and immediately sends it to its output channel. KP 0 sends the initial message (a single 32-bit integer) and measures the time that the message needs to make $m$ round-trips. In our case, we measure and report the duration of the composite [send $\rightarrow$ context switch $\rightarrow$ receive] transaction. Since the KPs perform no other work than receiving and sending a message, the performance of the benchmark is limited by the overheads of message-passing and scheduling.

#### 5.2.1.2. AES

The AES KPN is designed to investigate performance of a typical embarrassingly parallel workload. Embarrassingly parallel workloads can be divided into subtasks that can execute completely independently of each other.

[2]The software can be downloaded at `http://www.bitmover.com/lmbench`

The AES KPN (figure 5.4) consists of the source KP $S$ and $n$ worker KPs. The source KP hands out equally-sized chunks of memory to $n$ worker KPs, where each message contains a pointer to the beginning of the memory chunk and its length. When a worker receives a message, it encrypts the chunk with 128-bit AES algorithm in ECB mode and sends the reply message back to the source KP.[3] The source KP is the serial part of the AES benchmark, but its CPU usage is negligibly small compared to that of workers. We therefore expect that benchmark's performance will increase proportionally with the number CPUs used for execution.

The benchmark is configurable through the following parametrs:

- The base-2 logarithm $b$ of the total memory block size that will be encrypted.
- The base-2 logarithm $k$ of the chunk size.
- The number of encryption passes over each chunk.

A worker KP is created for each chunk, so there are $n = 2^{b-k}$ worker KPs in total.

#### 5.2.1.3. H.264 encoder

The goal of the H.264 benchmark is to investigate performance of irregular KPNs based on real-world applications. Our KPN representation of an H.264 video encoder (figure 5.5) is only a slight adaptation of the encoder block diagram found in [90], with each functional unit being implemented as a KP.

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.

Due to the complexity of developing a KPN-based implementation of an H.264 encoder, we have used an artificial workload consisting of loops that consume the amount of CPU time which would be used by a real codec on average. To gather this data, we have profiled x264 [91], an open-source H.264 encoder, with the *cachegrind* tool [81]. The profiling results are shown in table 5.5.

The benchmark program is configurable with the following parameters:

[3] ECB mode should never be used in production-level code. CTR mode, which *is* appropriate for production uses, is also trivially parallelizable. It has practically the same performance characteristics as ECB since it adds only a single operation – an integer increment – to the ECB mode.

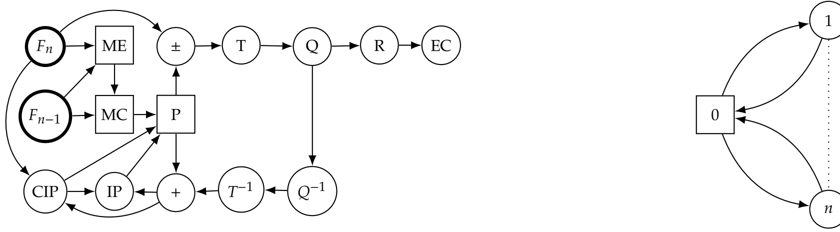



Figure 5.5.: H.264 block-diagram, adapted from the H.264 whitepaper [90]. Inputs to the codec are current and reference frames ($F_n$ and $F_{n-1}$). Since prediction (P), motion compensation (MC) and motion estimation (ME) blocks are together using over 50% of the total processing time and are perfectly parallelizable, we have parallelized each block by dividing the work over $n$ workers. The figure on the right exemplifies parallelization of a single block.

| Stage | Time % |
|---|---|
| Choose inter-/intra-prediction (CIP) | 8.08 |
| Intra-analysis (IP) | 5.02 |
| Inter-analysis (MC+ME) | 34.36 |
| Prediction (P) | 23.83 |
| Macroblock encoding (T,Ti,Q,Qi) | 8.69 |
| Entropy encoding (EC) | 6.72 |
| Other, including reordering (R) | 13.3 |

Table 5.5.: Cachegrind profiling data of x264, taken from [92].

- Amount of work performed for each input frame, so we can “encode” videos at different frame-rates.
- Number of encoded frames.
- Number of workers in *each* of the three parallel stages.

The results obtained with this benchmark are a good performance indicator for the implementation of *Nornir itself*. However, the artificial workload ignores other factors, such as memory bandwidth usage and variation in processing time across frames, so it cannot be used as a performance predictor for a real H.264 encoder implemented in the KPN framework.

#### 5.2.1.4. Random KPNs

Our motivation for using randomly generated KPNs, an example of which is shown in figure 5.6, as a workload is two-fold. First, we wanted to simulate complex topologies that might be generated by KPN generator tools [18, 17, 74]. Second, we wanted fine control over parallelism granularity.

| Value | $l$ | $u$ | $\overline{T_0}$ |
|---|---|---|---|
| Real time | 0.995662956 | 0.995982601 | 0.995822778 |
| CPU time | 0.995741274 | 0.995874908 | 0.995808091 |

Table 5.6.: CPU time used by $T_0 = 34500000$ iterations of the loop from figure 5.7 on machine described in section 5.1.1: 95% confidence interval $[l, u]$ and mean $\overline{T_0}$.

A random KPN consists 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 a uniform random number between one and the user-specified maximum number, while the degree of each node, and the number of layers between it and the target node are geometrically distributed random numbers. The KPN may have additional back-edges that create cycles, but each node is incident to at most one back-edge, be it outgoing or incoming. KPs exchange messages containing a single integer that tells the receiving KP how much CPU time it should use for that particular message. KPs use CPU time by using the `burn_cycles` function, shown in figure 5.7, supplying the received integer as the `niter` parameter. We have measured that $T_0 = 34500000$ iterations of the loop on machine described in section 5.1.1 uses approximately 1 second of CPU time; see table 5.6 for more detailed statistics.

Parallelism granularity is controlled by three user-specified parameters, $n$, $T$ and $d$, where $n$ is the number of messages, while $T$ and $d$ (*work division factor*) are scaling constants determining work granularity. The source KP sends $n$ messages, with each message representing $w = T/d$ CPU seconds; the CPU time is converted to the `niter` parameter of function `burn_cycles` by formula $wT_0$. If $n$ and $d$ are varied so that the ratio $n/d$ is held constant, the same amount of work is divided across a different number of messages, with higher values of $d$ representing finer parallelism granularity.

Every KP executes the following algorithm:

1. Set $t \leftarrow 0$.

2. Read a message $m$ from each of $n_i$ inputs and set $t \leftarrow t + [m]$, where $[m]$ denotes the contents of message $m$ (an integer).

3. $t$ now contains the sum of all integer values received from every input channel. Consume CPU time by calling `burn_cycles(t)`.

4. Send a single message containing $t/n_o$ units of work to each of $n_o$ outputs. If $t$ is not divisible by $n_o$, the remainder is added to the message being sent to the last channel.

5. If a KP has a back-edge, a message is sent or received, depending on the edge direction, along that channel.

Messages received along back-edges are discarded and do not contribute to the network's workload; their sole purpose is to generate more complex synchronization

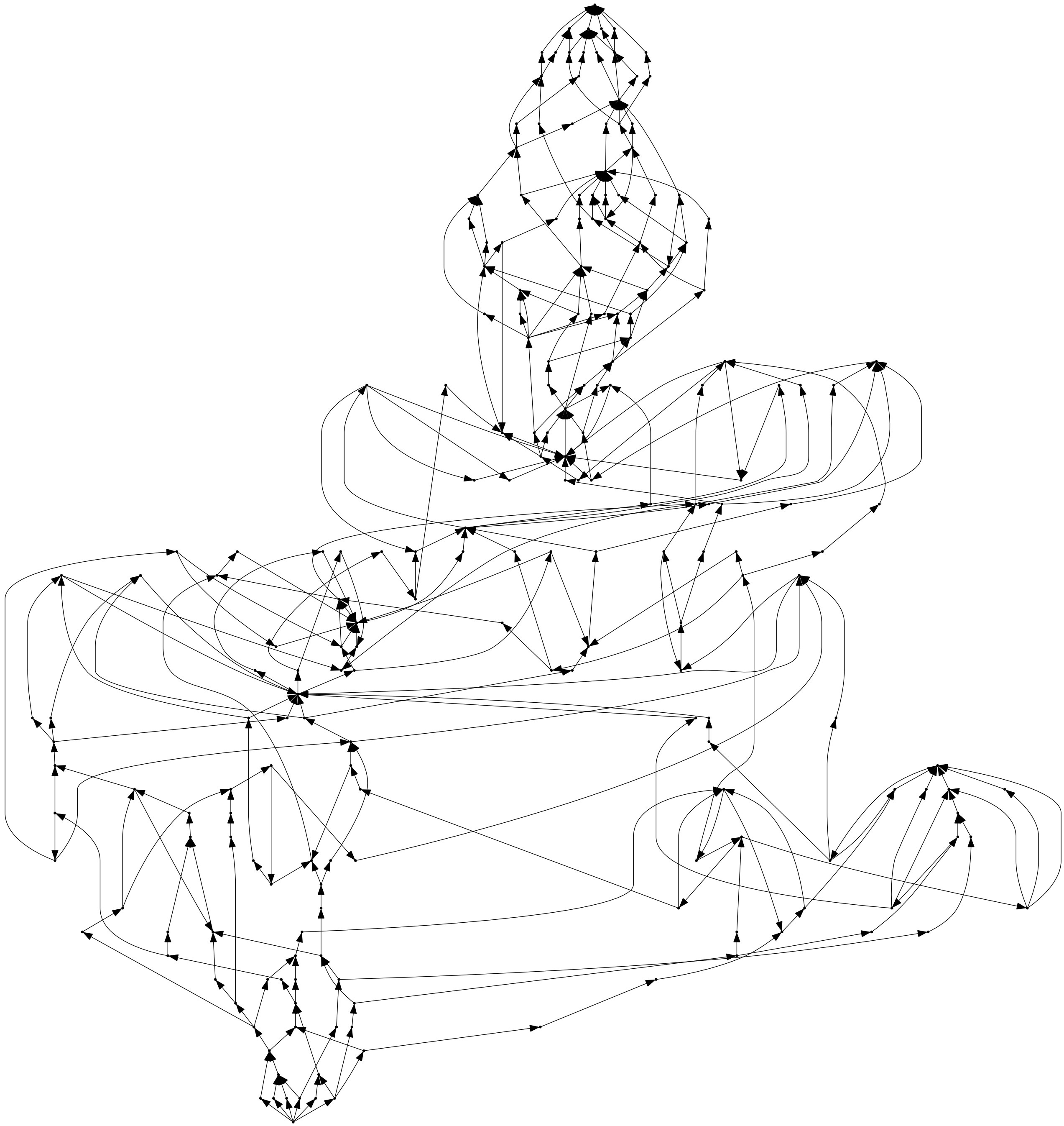


Figure 5.6.: A random graph with 212 nodes, 333 edges and 13 cycles. The source node is at the bottom, the sink node is at the top.

```
void burn_cycles(unsigned niter)
{
  volatile long a, b, tmp;

  if(b == 0)
    b = 17;
  while(niter--)
    tmp = a/b;
}
```

Figure 5.7.: Function that consumes amount of CPU time proportional to `niter`. Variables are deliberately used without initialization, and `volatile` prevents the compiler from optimizing out the loop.

patterns. This algorithm preserves the total amount of work in the network, so the workload $w$ contained in a single message sent by the source KP will equal the value $w$ received by the sink KP. It is also worth noting that because of branching in the network, many individual KPs will perform *less* than $w$ units of work per message.

#### 5.2.1.5. Pipeline

As random KPN, a pipeline consists of a source KP, a sink KP and a number of KPs in between. Each KP, except source and sink, have one input and one output, and they are are connected in a simple pipeline, resembling that of figure 3.9. The workload is generated in the same way as for the random KPNs: the source KP sends $n$ messages, each representing $T/d$ CPU seconds. Since there is no branching in the pipeline graph, the work division factor $d$ has an intuitive interpretation in this case: each KP in the network uses exactly $T/d$ CPU seconds for each received message.

### 5.2.2. Performance of implementation options

In section 2.2, we have reviewed OS mechanisms for scheduling, message-passing, deadlock-detection and synchronization. The former three are the main ingredients of a KPN run-time and the latter is a necessary implementation tool. While the native scheduling and synchronization mechanisms are functionally satisfactory, the native message-passing mechanisms have other problems besides unreliable support for deadlock detection. One problem is that they are a system-wide limited resource, so it would not be feasible to use, for example, a separate pipe for each channel in a large KPN. One could possibly multiplex several channels over a single OS resource, but this would significantly complicate and slow down the implementation. Another problem is that all system-provided communication mechanisms, except for possibly POSIX message queues, require a system call for every send and receive operation, which incurs a performance penalty, especially for small messages. We have decided to compare the following orthogonal implementation options:

- Our optimized m:n scheduling mechanism with static assignment of KPs to CPUs (see section 4.2.1) against the OS's native 1:1 thread scheduler.
- Our optimized zero-copy message-passing mechanism against POSIX message queues (native transport).

In this evaluation, we have used the ring, AES and H.264 benchmarks.

#### 5.2.2.1. Methodology

The experiments have been run on a 64-bit, dual-core AMD Athlon on 2.2 GHz with 2 GB of RAM running Solaris 10. The benchmarks were compiled with Sun's C++ compiler (version 5.9, patch 124864-07 2008/08/22) for 64-bit mode and with maximum optimizations turned on (`-m64 -fast`). All reported results are based on 10 consecutive runs. In all configurations, collection of detailed accounting data was *disabled*, and default channel capacity has been set to 64 messages. In this early set of experiments, deadlock detection was *enabled*, but it was only partly implemented, so its overheads had very little impact on measurements.

The main performance metric in our evaluations is the *real (wall-clock) time* measured by a high-resolution timer. 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 we do not use them for evaluation because they do not reflect the reduced running time with multiple CPUs, and because they do not take into account sleep time, which significantly impacts the real running time.

#### 5.2.2.2. Results

The plots and discussions below are based on the average of 10 measurements. We have also computed the relative error of our measurements according to the formula $\epsilon_r = \frac{M-m}{\mu}$, where $M$ is the maximum running time, $m$ the minimum running time and $\mu$ is the average running time for each set of 10 measurements.

**Ring** In the ring benchmark, only a single message is traveling through the network, so every communication requires a context switch and synchronization with the next KP in the ring. Because of frequent synchronization and context switching, the benchmark is very sensitive to the assignment of KPs to CPUs. In these experiments, a *good* assignment requires the least amount of inter-CPU synchronization and is achieved by distributing $n$ KPs into as many roughly equal groups as there are CPUs (say $k$) and assigning KPs in the same group to the same CPU. For example, KPs $0, 1, \ldots, \lfloor n/k \rfloor$ shall be assigned to CPU 0. A *bad* assignment requires inter-CPU synchronization for every transaction and is achieved by assigning KP $i$ to CPU $i \bmod k$. With this benchmark, we evaluate the impact of the following factors on performance:

- Choice of the message transport mechanism: our optimized transport vs native transport.
- Choice of the scheduling mechanism: our optimized m:n scheduler vs. the native 1:1 scheduler.
- Assignment of KPs to CPUs (good vs. bad).

The benchmark has been run for the number of KPs $n$ varying from 10 to 1000 in steps of 50. The number of round-trips $m$ is set to $m = \lfloor 10^6/n \rfloor$ so that the total number of [send $\rightarrow$ context switch $\rightarrow$ receive] transactions is constant and equal to $10^6$. These settings give a sufficiently long execution time and number of transactions to cancel out the the noise of scheduling effects such as interrupt processing. Since dynamic memory (de)allocation is an integral part of zero-copy message-passing, its costs are included in the presented results. The largest relative error over all experiments is $\epsilon_r = 0.1$.

Figure 5.8 shows the results of benchmarking three different configurations: native scheduler with native transport (top), native scheduler with optimized transport (middle) and optimized scheduler with optimized transport (bottom). From the figure we can draw several conclusions.

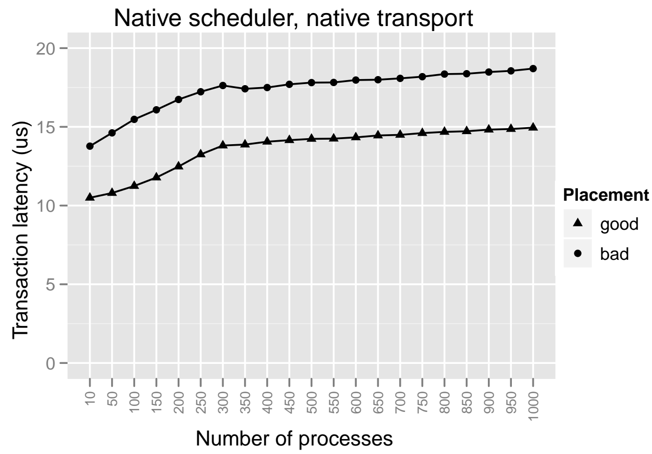
First, inter-CPU synchronization is expensive, which is confirmed by observing the difference in running times of the benchmark with good and bad KP placement. We believe that this is also the cause for the anomaly that can be seen in the bottom plot of figure 5.8, where the running time *increases* as the number of KPs *decreases* below 100.

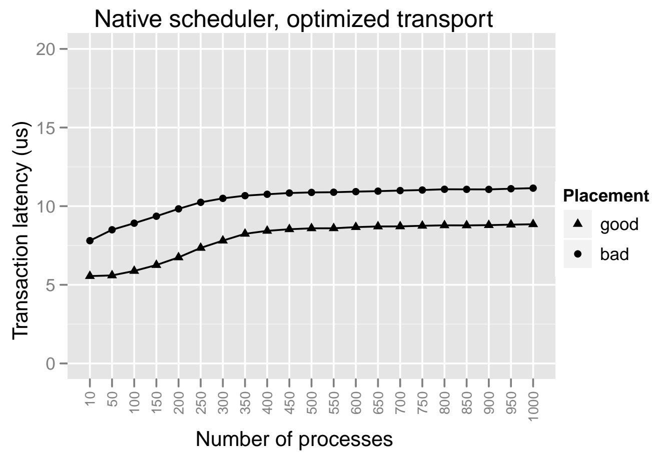
Second, the optimized transport is $\sim$ 1.7–1.9 times faster than the native transport; the exact speedup is dependent on the workload, KP placement and message size. As can be seen from figure 5.9, the copying semantics of the native transport causes it to degrade in performance as message size grows above 256 bytes.

Third, our user scheduler performs best with a single runner because there is no contention over channel locks and delays caused by sleeping on the semaphore. In the configuration with optimized scheduler using two runners, optimized message transport, and bad placement, it performs slightly worse than the native scheduler for less than $\sim$ 100 KPs (compare the middle and bottom plot in figure 5.8). As the number of KPs grows, it starts to outperform the native scheduler, whose performance becomes worse while that of the optimized scheduler remains constant, and 1.25 times faster for $n = 1000$ KPs. With good KP placement and $n = 1000$, our scheduler is 6.5 times faster than the native scheduler, and achieves nearly the same performance as with a single runner.

The last experiment (see figure 5.10) explores the optimal number of runners for the optimized scheduler under load of 1000 KPs with good and bad placement. As expected, the best running times are achieved for one and two runners since the machine has two cores.

**AES** One of the requirements for our run-time is to enable overdecomposition without incurring large performance penalty on compute-intensive tasks. We have thus used the AES benchmark to determine how well Nornir supports overdecomposed applications.



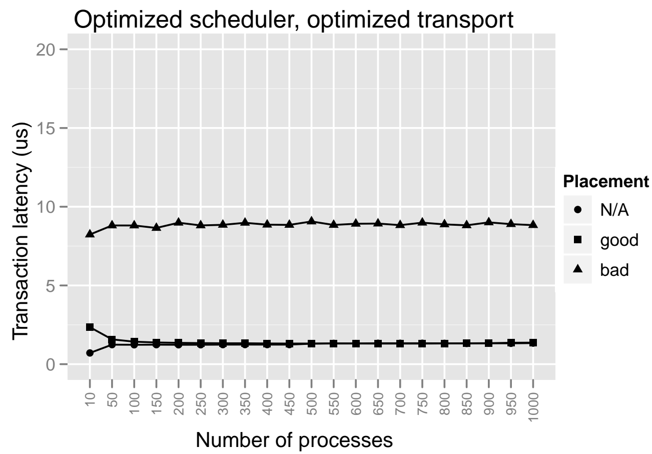




Figure 5.8.: Ring benchmark with different Nornir configurations and KP placement. The "N/A" experiment has been run with only 1 runner.

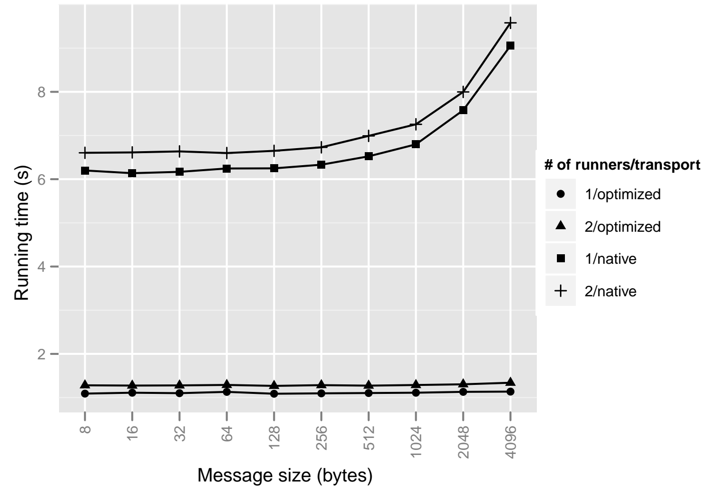



Figure 5.9.: Ring benchmark: message transport performance vs. message size. Optimized scheduler using 1 and 2 runners, 100 KPs and 10000 round-trips ($10^6$ transactions, as in other experiments), good placement.

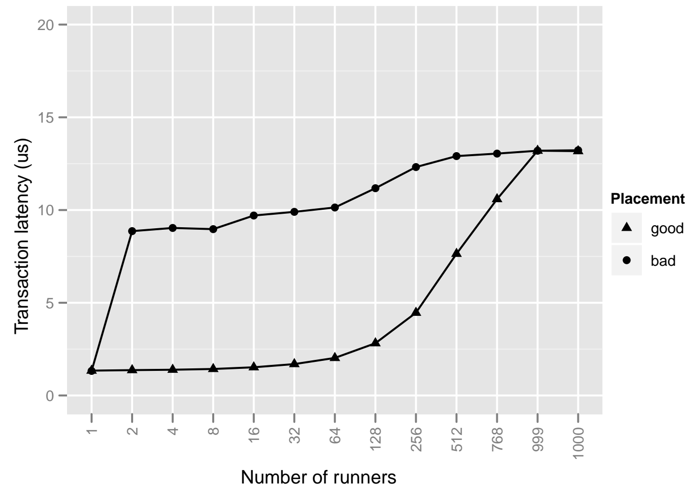



Figure 5.10.: Running time of the ring benchmark with 1000 KPs and 1000 round-trips under the optimized scheduler and transport with varying number of runners. *The scale on x-axis is not uniform.*

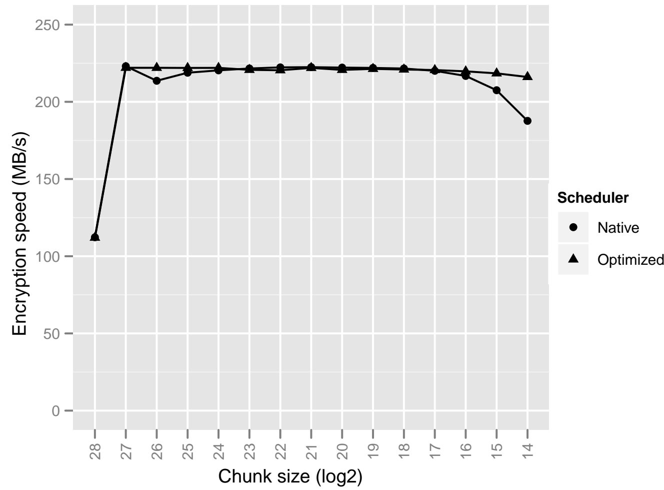



Figure 5.11.: Performance of AES encryption with native and optimized (2 runners) scheduler. The number of workers increases to the right, as the chunk size decreases.

We have fixed the total memory block size to $2^{28}$ bytes (256 MB, to avoid possible swapping), and the number of encryption passes over each chunk to 8 (to achieve sufficiently long running time). We have varied the base-2 logarithm of memory chunk size, $k$, from 14 to 28. Since a worker KP is created for every chunk of memory, this corresponds to $n = 2^{28-k}$ KP workers in total. In the U/2/O configuration with optimized transport and scheduler with 2 runners, KPs are assigned to CPUs in a round-robin manner so that load is evenly distributed among CPUs.

Figure 5.11 shows the encryption throughput versus the number of workers; note that the *number of workers increases* to the right, as the *chunk size decreases*. The 50% drop in speed at $n = 28$ occurs because the task is serially executed on a single CPU. As the number of KPs increases beyond 2048 (chunk size less than $2^{17}$), the encryption speed decreases, but much more sharply for the native scheduler. To ensure that this was not caused by swapping, we monitored the system activity during the benchmark run with the `iostat` tool, which reported no disk activity. We have observed the largest relative error $\epsilon_r = 0.06$.

Furthermore, when using the native scheduler with 16384 worker KPs, there are approximately 25000 context switches, 35000 system calls and 30000 instruction TLB misses per second.[4] In contrast, the optimized scheduler with 16384 worker KPs has $< 500$ system calls and context switches per second (which almost coincides with the numbers on an idle system), and $< 2000$ instruction TLB misses per second ($< 1000$ on an idle system). While we have not been able to find the exact causes of this behavior, we believe, based on real-time monitoring by *dtrace* and *truss*, that they stem from the combination of the (preemptive) native scheduler and lock contention. Whatever the

[4] We have used the `cputrack` program to program the CPU's performance counters and report the results.

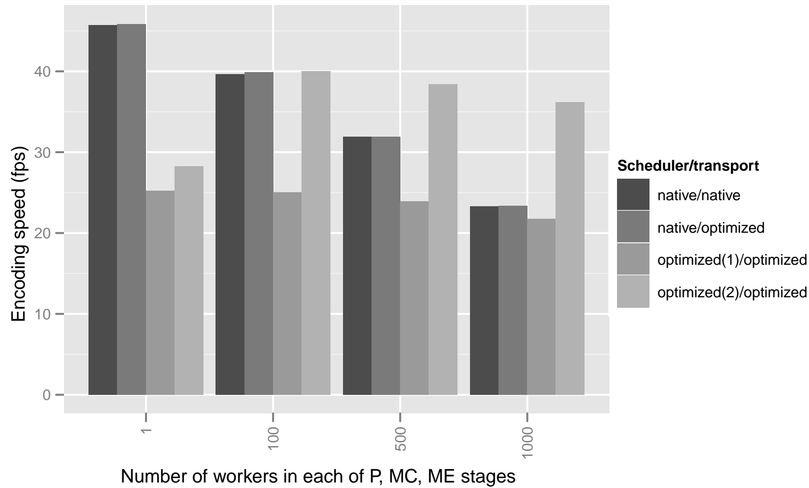



Figure 5.12.: Performance of the H.264 encoder with native and optimized schedulers and transport. Experiments with optimized scheduler used 1 and 2 runners.

root cause, it is obvious that the native scheduler has some trouble as the number of threads increases beyond 2048, which indicates that our scheduler is better suited for executing overdecomposed problems than the native OS scheduler.

**H.264** We have configured the H.264 benchmark to "encoding" speed of approximately 25 fps in real-time and measured the time needed to "encode" 300 frames (12 seconds) in different Nornir configurations. For each configuration, the benchmark is executed with 1, 100, 500 and 1000 workers in each of the P, MC and ME stages (up to 3000 additional workers in total). The largest observed relative error is $\epsilon_r = 0.03$.

H.264 is computation-intensive benchmark with little communication in our implementation. From the results in figure 5.12 we can conclude that:

- Data dependencies in KPN prohibit linear scalability with the number of cores.
- Message transport (optimized vs. native on optimized scheduler) has no influence on running time because the workload is CPU-bound and few messages are exchanged.
- With few KPs and little available parallelism, the native scheduler's load-balancing outperforms our static assignment of KPs
- When the application is overdecomposed and KPs are carefully placed, our scheduler performs as well as, or better than the native scheduler.

#### 5.2.2.3. Discussion

We have evaluated performance of message-passing and scheduling mechanisms in their native (POSIX message queues and the native 1:1 scheduler) and optimized implementation. Experiments using the ring, AES and H.264 benchmarks show that our implementation makes significant scalability and performance improvements over native mechanisms: in our tests, message transport is 1.7–1.9 times faster, and the scheduler is up to 6.5 times faster.

Figures 5.11 and 5.12 show that the optimized scheduler scales much better with the number of KPs than the native scheduler, even when the KPs are CPU-intensive. When the number of workers in each of the P, MC and ME stages of the H.264 benchmark is increased from 1 to 1000, our scheduler suffers a performance drop of $\sim 10\%$, while the performance of the native scheduler is nearly halved.

However, the H.264 benchmark also shows that static assignment of KPs to CPUs is not enough and that lack of load-balancing mechanisms hurts performance in situations with few workers. In benchmark configuration with one worker, the native scheduler achieved more than 50% higher "encoding" speed than with the optimized scheduler.

The main conclusion is thus that the m:n scheduling model is promising because it has much lower overheads than the 1:1 model, but that it also needs to be supplemented with dynamic load-balancing algorithms to be able to compete with performance of the native scheduler.

### 5.2.3. Performance evaluation of Nornir

In the previous set of experiments, we have evaluated performance of various KPN implementation options. We have found that the m:n scheduler combined with the optimized zero-copy transport gives the best performance on many tasks, but that it also needs dynamic load-balancing in order to be able to compete with the 1:1 scheduler on irregular workloads. The experiments have also shown that the cost of message copying is negligible as long as message size is less than 256 bytes.

Based on these findings, we have implemented two dynamic load-balancing methods (work-stealing and a method based on graph partitioning, both described in section 4.2, and we have replaced the zero-copy message transport with message copying. The latter change eliminates the costs of dynamic memory (de)allocation on every receive or send operation, which are higher than the costs of copying when messages are small. Lastly, we have also fully implemented run-time deadlock detection and resolution algorithm. For our tests, we have used the AES, H.264, random graph and pipeline benchmarks. In evaluations presented in this section, we have only used the work-stealing scheduler. Comparative evaluation of work-stealing and graph partitioning load-balancing methods is the topic of section 5.3.

#### 5.2.3.1. Methodology

Nornir has been configured to use the m:n work-stealing scheduler with both deadlock detection and detailed accounting *enabled* and a default channel capacity of 64 messages. The benchmarks have been compiled and run on the same hardware and OS as described at the beginning of section 5.1.1. As in the previous experiment set, our conclusions are based on the average results of 10 consecutive runs.

**Metrics** As previously explained, an important metric is real (wall-clock) running time. We have run benchmarks on 1, 2, 4, 6 and 8 CPUs, and for each number of CPUs we have computed the speedup over 1 CPU. A speedup of an embarrassingly parallel benchmark, such as AES or pipeline, that is significantly less than the number of CPUs allocated to Nornir would indicate a possible performance problem in Nornir. In the current Nornir implementation, the main potential source of possible scalability problems is the centralized deadlock detection algorithm which starts whenever a KP blocks on read or write. We have thus investigated the impact of *deadlock detection rate*, i.e., the number of started deadlock detections per second, on speedup.

**NUMA issues** Since a context switch also switches stacks, it can be expected that cached stack data will be quickly lost from CPU caches when there are many KPs in the network. We have measured that the cost of re-filling the CPU cache through random accesses increases by $\sim 10\%$ for each additional hop on our machine (see appendix C for pertinent details about computer architecture). Due to an implementation detail of Nornir and Linux's default memory allocation policy, which first tries to allocate physical memory from the same node from which the request came, all stack memory would end up being allocated on a single node. Consequently, context switch cost would depend on the node a KP is scheduled on. To average out these effects, we have used the `numactl` utility to run benchmarks under the interleave NUMA (non-uniform memory access) policy, which allocates physical memory pages from CPU nodes in round-robin manner.

**Benchmark parameters** The **ring** benchmark was configured with 1000 KPs and 1000 round-trips, totaling $10^6$ [send → context switch → receive] transactions.

The **H.264** benchmark was configured to "encode" 30 frames at rate of 1 frame per second, with the number of workers in each parallelized stage varying from 1. . . 512 in successive powers of two.

For **AES**, the total block size was set to $2^{28}$ bytes (256 MB), and the chunk given to each individual worker varied from to $2^{17}$ to $2^{28}$. In this benchmark, *smaller* chunks correspond to a *larger* number of workers, since each worker operates on its own chunk. Thus, as chunk size *increases* from 17 to 28, the number of workers *decreases* from 2048 to 1. The number of encryption passes over each chunk was set to 40.

We generated two random KPNs: the **RND-A** graph is a random graph consisting of 238 nodes and 364 edges in 50 layers; the **RND-B** graph is a random graph consisting of 212 nodes, 333 edges in 50 layers and 13 cycles; the **pipeline** graph consists of 50 stages.

For all three graphs, we have set $T = 1$ and varied the work division factor $d$ from 1 to 90000 (in unequal steps). We have set $n = d$, so the total amount of work generated by the source was constant. Thus, 1 second of CPU time was split into up to 90000 messages, each carrying workload of 1/90000 CPU seconds. Because of branching in the network, individual KPs in RND-A and RND-B networks may receive even smaller work amounts.

#### 5.2.3.2. Results

For each data point we have also investigated the relative error, which we have computed according to the formula $\epsilon_r = \frac{u-l}{2l}$, where $u$ and $l$ are the upper and lower bound of the 95% confidence interval calculated by using the Student's t-distribution with 9 degrees of freedom. We have computed the *maximum* $\epsilon_r$ over all combinations of work-divisions and number of CPUs and found it to be rather small: 12% for the RND-B benchmark and around 5% for other benchmarks except RND-A. Some experiments of the RND-A benchmark have a large relative error: up to 88% on 4 CPUs and $d = 90000$. For completeness, we show the box-and-whisekers plot[5] of all RND-A running times in figure 5.13. The figure reveals three general trends:

- Variance exhibits a sudden increase at a certain threshold value of $d$.
- The threshold value lowers as the number of CPUs increases.
- The variance increases with the running time.

**General performance characteristics** We have first run the **ring** benchmark only on 1 CPU to evaluate the performance of context switch and accounting mechanisms. Table 5.7 shows 95% confidence intervals and relative error of the running time with accounting enabled and disabled. The results show that the overheads of detailed accounting almost double the cost of a single [send → context switch → receive] transaction for reasons described in section 4.5.2. Considering only mean values, Nornir running on a single CPU can perform $\sim 1.2 \cdot 10^6$ transactions with accounting disabled, and $\sim 7 \cdot 10^5$ transactions with accounting enabled. On multiple CPUs, the ring benchmark results depend heavily on the number of CPUs and scheduling policy, a topic investigated in detail in section 5.3. Put in perspective with existing user-mode schedulers, Saha et al. [83] have measured that their system uses $0.748\mu$s for context switch on 2.8GHz Xeon CPU. Our mean time is slightly higher because our machine is a bit slower (2.6GHz), and because our transaction time also includes the cost of sending and receiving a single 4-byte message.

Figures 5.14 to 5.18 show running time, speedup and deadlock detection rate on 1, 2, 4, 6 and 8 CPUs for the other five benchmarks. The best achieved speedups on 8 CPUs

[5] This is a standard way to show the distribution of a data set. The box's span is from the lower to the upper quartile, with the middle bar denoting the median. Whiskers extend from the box to the lowest and highest measurements that are not outliers. Points denote *outliers*, i.e., measurements that are more than 1.5 times the box's height below the lower or above the upper quartile.

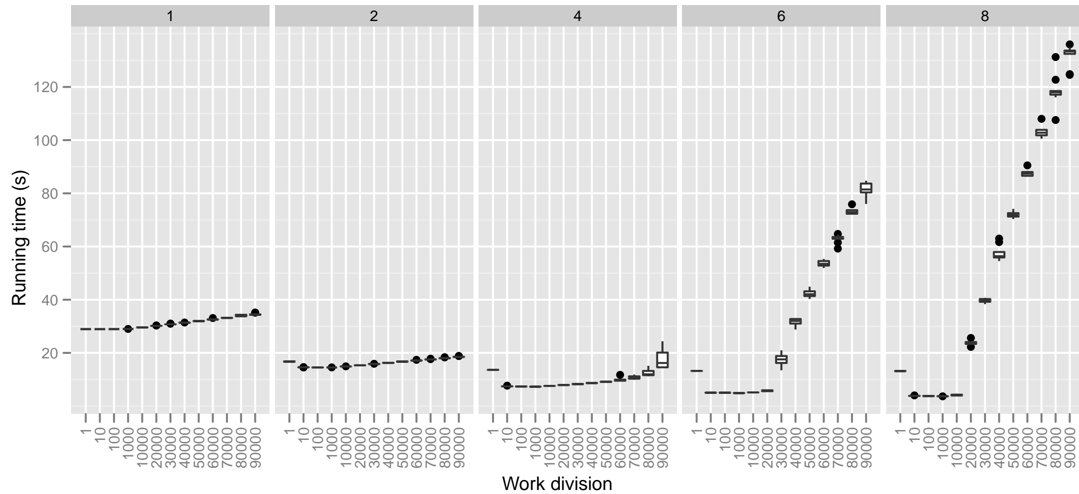



Figure 5.13.: Distribution of RND-A running times on 1, 2, 4, 6, 8 CPUs.

| Accounting | Mean | Low | High | $\epsilon_r$ (%) |
|---|---|---|---|---|
| Disabled | 0.7979 | 0.6205 | 0.9752 | 28.58 |
| Enabled | 1.4238 | 1.3649 | 1.4826 | 4.31 |

Table 5.7.: Transaction time (in microseconds) for ring benchmark with 1000 workers and 1000 round-trips with accounting enabled or disabled ($10^6$ transactions): mean, confidence intervals and relative error of running time at 95% level. The results are based on 10 measurements.

for pipeline, AES and H.264 benchmarks are 7.95, 7.46 and 2.73, respectively. All three benchmarks share the same overall performance characteristics:

- The running time decreases with increasing number of runners.
- The speedup increases with increasing work division $d$, and remains constant after $d$ has passed a certain value.

**Pipeline** exhibits a sharp speedup for $d = 10$ (see figure 5.14), as this is already greater than the number of CPUs in all experiments. Speedup continues to increase slightly up to $d = 10000$ where it reaches its peak value of 7.95 on 8 CPUs, and starts decreasing slightly for $d > 20000$. The slight increase in speedup as $d$ increases from 10 to 10000 comes from better pipeline utilization; only at $d = 10000$ are all channels in the pipeline are full most of the time. This decreases the frequency with which KPs block, which in turn decreases contention over run-queues. The impact of the latter factor is further investigated in section 5.3.2. At $d \geq 30000$, the amount of useful computation per message continues to decrease, while the number of sent messages increases proportionally. The increase in the number of messages increases the frequency of deadlock detection, while the simultaneous decrease in the amount of computation per message reduces the

opportunity of overlapping useful computation with deadlock detection. The centralized deadlock detection algorithm thus starts being a serialization point, which affects the speedup.

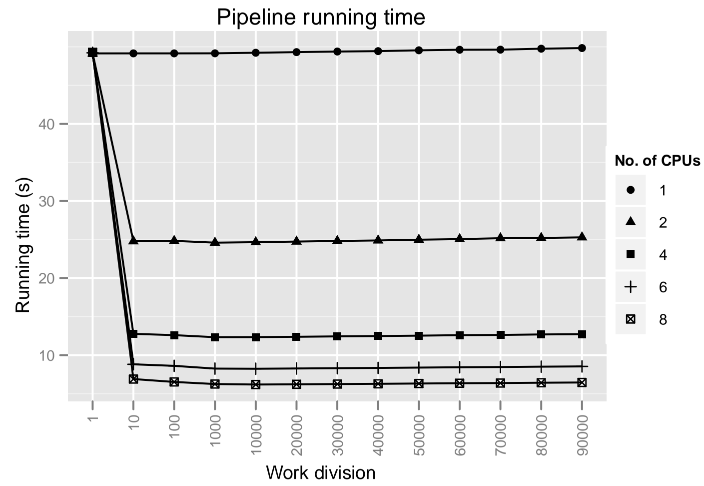
The **AES** speedup (see figure 5.15) reaches its peak when there are at least as many chunks as runners, and remains constant up to 2048 workers. Speedup on 6 CPUs is irregular and reaches its peak speedup at 256 workers (chunk size $2^{20}$). Even though AES is perfectly parallelizable, the peak speedup on 8 CPUs is 7.46, which is the smallest speedup on 8 CPUs of all other benchmarks except H.264. However, AES is significantly different in one way from other benchmarks in that it is both CPU and memory-intensive. Given that it has been run under round-robin NUMA policy, we believe that the speedup is limited by the machine's memory architecture.

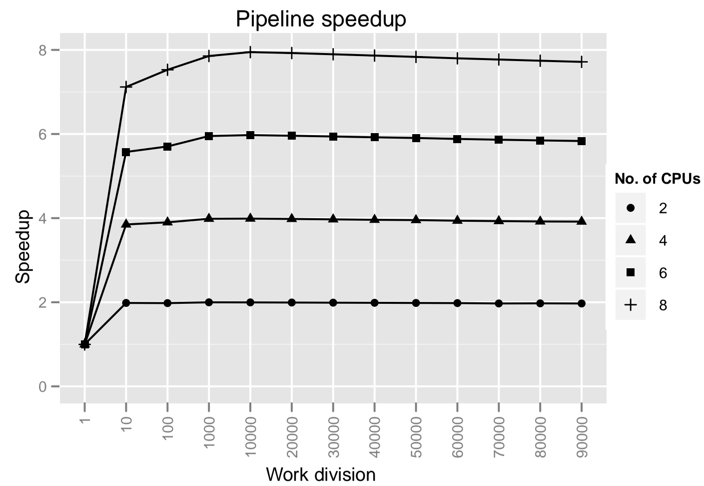
**H.264** has a similar performance profile (see figure 5.16) as AES, including the more irregular behavior on 6 CPUs, except that the peak achieved speedup on 8 CPUs is 2.73. In this case, the limitation stems from data-dependencies in network, which limit the available parallelism and, consequently, possible speedup.

Figures 5.17 and 5.18 show performance characteristics of the two random KPNs. Both plots have a similar profile: speedup grows up to a certain value of $d$ (1000 for **RND-A**, 100 for **RND-B**). The performance increases with increasing $d$ because more messages flow in the network, which increases parallelism. The peak achieved speedup for the two graphs is 7.86 for RND-A and 7.9 for RND-B. When $d$ increases beyond the optimal value, the speedup drops sharply at $d = 20000$ for RND-A and $d = 10000$ for RND-B. An exception is RND-A on 2 CPUs where speedup remains constant as $d$ increases and on 4 CPUs where speedup drops gradually. The sharp drop in performance is caused by the centralized deadlock detection mechanism, the overheads of which we investigate further in this section.

**Communication overheads** In all previous benchmarks, the communicated messages have been rather small (less than 32 bytes). We have used the same 50-stage **pipeline** to study the impact of message size on performance. As previously, $d$ messages have been generated by the source KP, each containing $1/d$ seconds of CPU time. A noticable slow-down (see figure 5.19) happens regardless of message size and only at $d = 10^5$, which is equivalent to 10 $\mu s$ of work per message, which is only 7 times greater than the time needed for a single [send → context switch → receive] transaction. As expected, the drop in performance at $d = 10^5$ is proportional with message size and the number of sent messages. Somewhat more unexpectedly, the drop is also proportional with the number of CPUs. Larger message sizes take more time to copy, which causes greater contention over channel locks with increasing number of CPUs. This hypothesis is confirmed by the spin rate, a value that is proportional with the time spent waiting on channel locks.[6]

[6]Actually, it is proportional to the sum of spins on channel locks and spins on the deadlock detection lock. Both graphs have the same profile, and the peak deadlock detection rate is ~ 15000 per second.



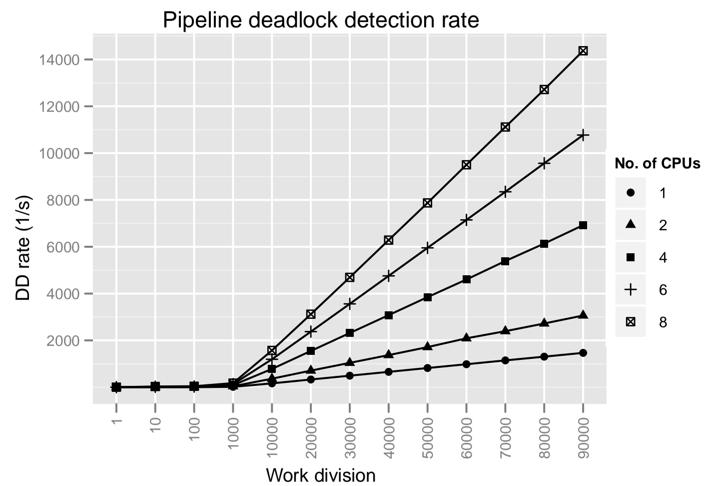
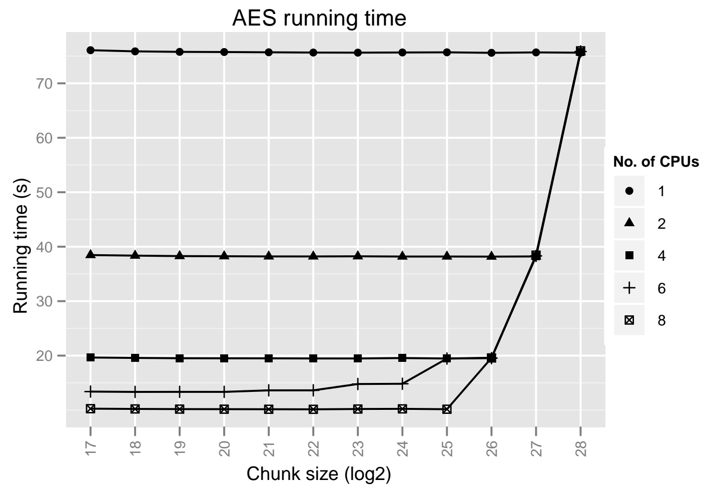
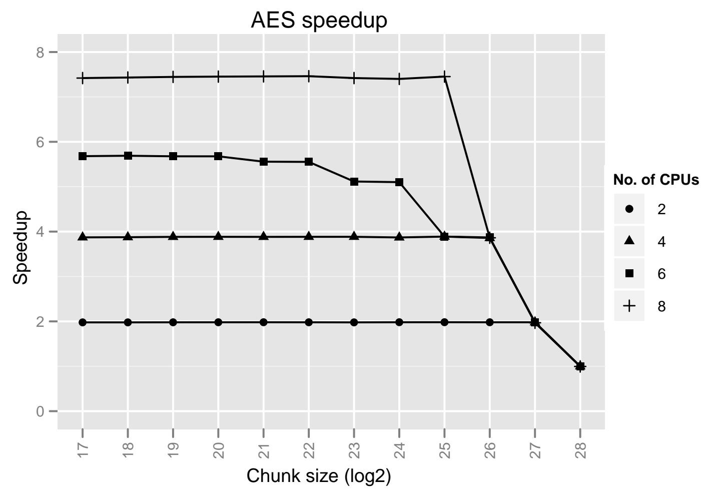




Figure 5.14.: Pipe performance characteristics.



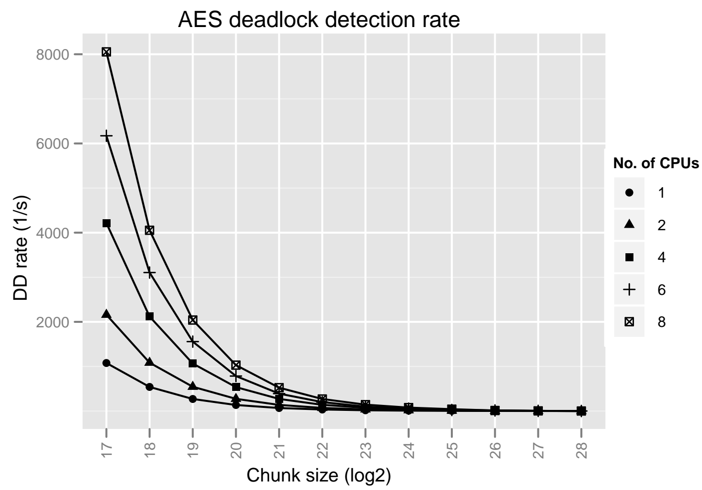
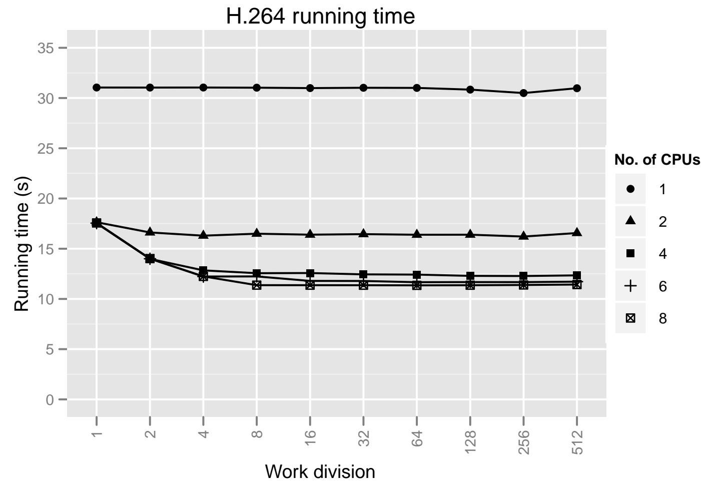
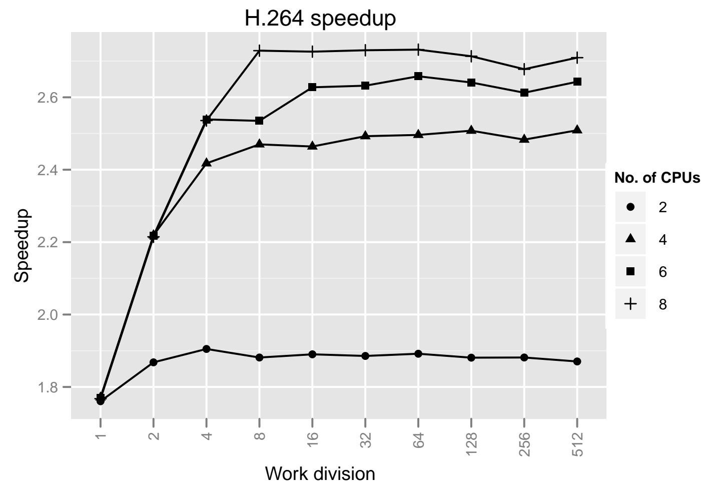




Figure 5.15.: AES performance characteristics.



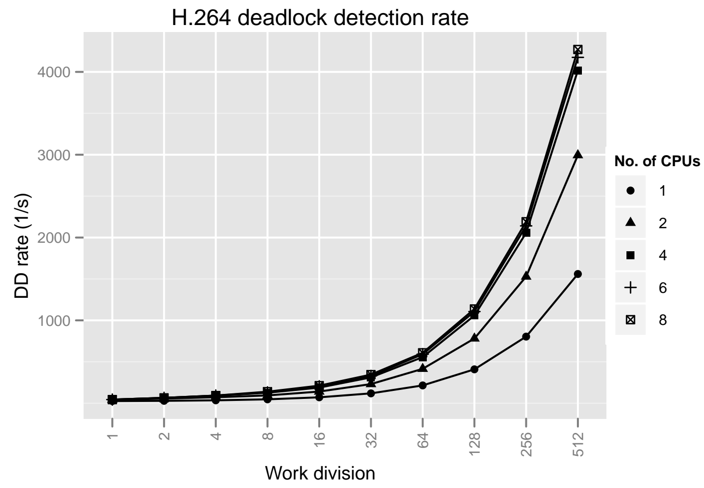
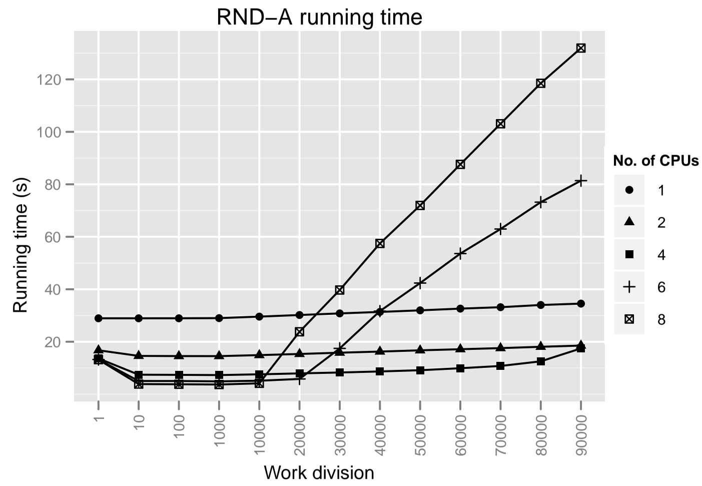
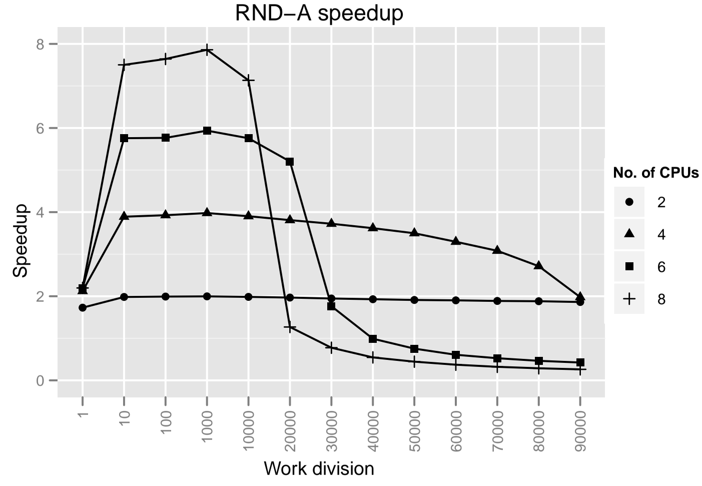




Figure 5.16.: H.264 performance characteristics.



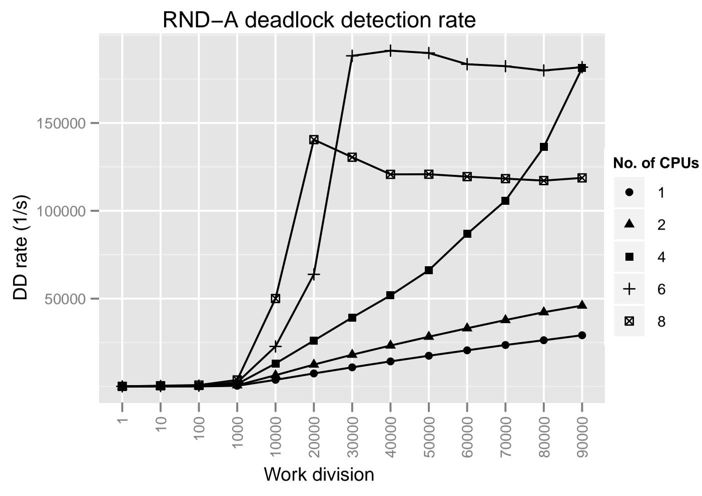
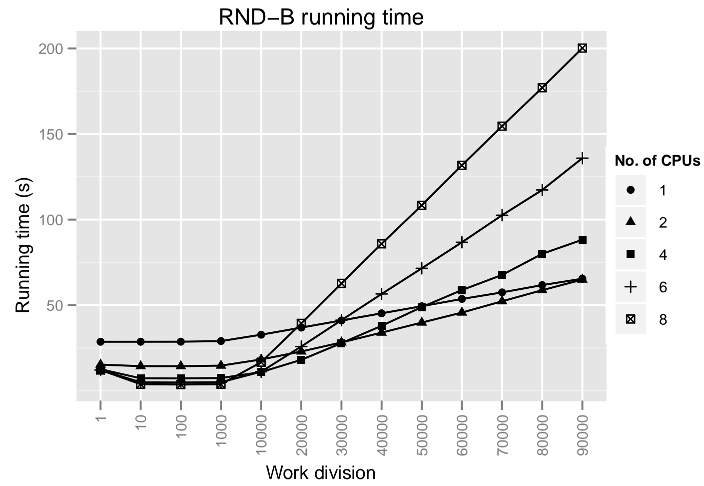
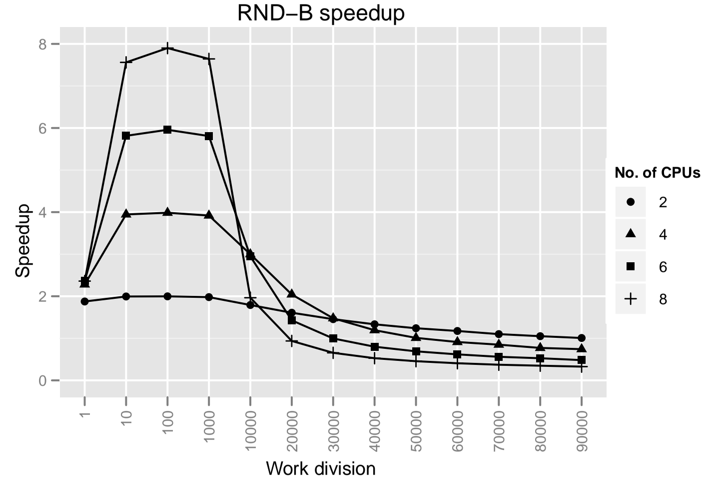




Figure 5.17.: RND-A performance characteristics.



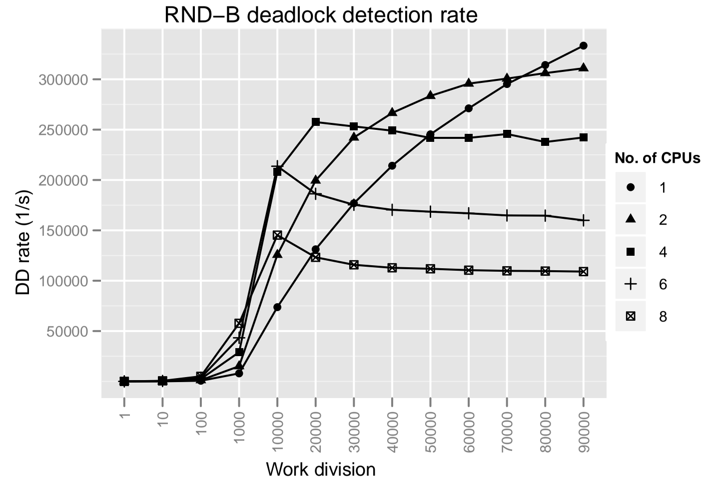




Figure 5.18.: RND-B performance characteristics.

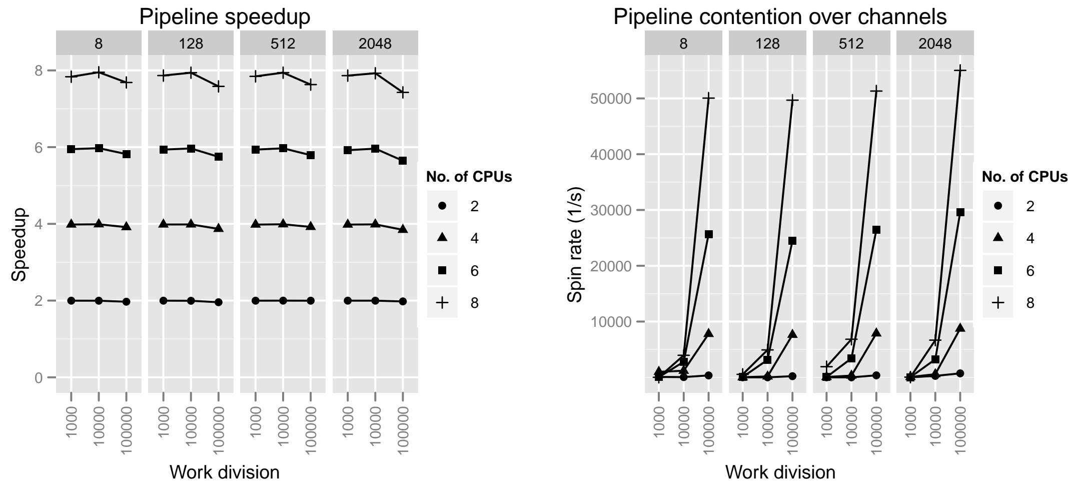



Figure 5.19.: Pipeline benchmark: effect of different message sizes (8,...,2048) bytes across three different work divisions (*d*).

**Deadlock detection overheads** Figures 5.14 to 5.18 also show the average number of started deadlock detections per second for each of the five experiments. However, a deadlock has actually never occurred in any of our experiments. The graphs have two common features: the deadlock detection rate is proportional with work division (consequently, also with the number of messages) and the number of CPUs. However, they differ greatly in scale: the peak deadlock detection rate ~ 14000 for pipeline, ~ 8000 for AES, ~ 4000 for H.264, ~ 200000 for RND-A and ~ 300000 for RND-B.

The first three and the last two workloads differ in one key aspect. The deadlock detection rate grows *linearly* with work division for the pipeline, AES and H.264 graphs, and the RND-A graph on up to 2 CPUs. However, it exhibits a *superlinear* growth for the RND-A graph on 4 CPUs and *sudden jump* on RND-A on 6 and 8 CPUs, as well as on the RND-B graph for any number of CPUs. After the abrupt increase, the deadlock detetction rate becomes quickly saturated and is *inversely* proportional with the number of CPUs, which indicates that the centralized data structure causes scalability problems. However, as can be seen from the figures, the central lock impacts performance only when the deadlock detection rate becomes greater than ~ 50000 started deadlock detections per second.

#### 5.2.3.3. Discussion

We have used synthetic benchmarks to evaluate application performance and scalability of Nornir. The benchmarks have been run on 1, 2, 4, 6 and 8 CPUs with work-stealing scheduling algorithm and deadlock detection and accounting enabled. We have also varied the work division factor *d*, which influences the total traffic volume and, in some cases, also the number of KPs in the network.

All benchmark programs, except H.264, have achieved a nearly linear speedup as the number of CPUs has been increased. The H.264 benchmark achieved maximal speedup of 2.73 on 8 CPUs, which is a consequence of limited parallelism available in the network. The random graph benchmarks have also achieved a nearly 8-fold speedup on 8 CPUs, but only up to a certain threshold value of $d$, above which the performance dropped drastically; in certain cases the performance was worse than on 1 CPU. We have determined that this degradation in performance is caused by frequent runs of the deadlock detection algorithm: the performance on 8 CPUs drops when the number of deadlock detections exceeds $\sim$ 50000 per second. The other three benchmarks have not suffered any scalability problems despite the fact that they too had a rather high deadlock detection rate on 8 CPUs: $\sim$ 14000 for pipeline, $\sim$ 8000 for AES and $\sim$ 4000 for H.264.

We have also used the 50-stage pipeline to evaluate overheads of the copying message-passing mechanism. We have found that scheduling overheads become visible when the processing time per message drops below $\sim 10\mu$s which is only 7 times greater than the cost of a single transaction. Furthermore, message sizes up to 2048 bytes are handled very efficiently: in the worst-case, the speedup with 2048-byte message, $d = 100000$ and 8 CPUs, is only $\sim$ 3.4% smaller than speedup for 8-byte messages and same $d$.

### 5.2.4. Conclusions

Our first set of experiments has shown that our message-passing and m:n scheduling mechanisms have better performance and scalability than POSIX message queues and the native 1:1 scheduler. Currently, the only advantage of the 1:1 scheduler seems to be that it is friendlier towards other applications running on the same machine: when there are fewer ready KPs than available CPUs, the kernel's scheduler will automatically free those CPUs for other applications. The m:n scheduler, on the other hand, loops until a KP appears in its run-queue, which unnecessarily wastes CPU resources when an application is in a phase where its average parallelism is less than the number of runners. This problem can be addressed by adapting a dynamic parallelism estimation mechanism [93] to Nornir's cooperative scheduling mechanism.

Our second set of experiments has shown that Nornir scales well up to 8 CPUs, despite our choice to use mutexes for protecting the scheduler's run-queues, which we have made to simplify our implementation. The benchmarks have also revealed that centralized deadlock detection can severely hurt performance, which was manifested only with the random graph benchmarks. An easy solution for this problem is to select a higher default channel capacity for problematic work-loads (all benchmarks were run with capacity of 64 messages), which will reduce the frequency of deadlock detections, but may also increase memory consumption beyond acceptable levels.[7] A better, but harder, solution is to extend Nornir with a distributed deadlock detection and

[7]When all channels are initialized to default capacity of $D$ messages, the memory space reserved for channel buffers is equal to $\sum_C D \cdot s(C)$ where $C$ runs over all channels and $s(C)$ is the size of single message carried by the channel.

resolution algorithm, such as the one described in [22]. We suggest two other possible improvements for the current implementation:

- Replacing channel's busy-waiting mutexes with a refined locking strategy, or by using single-producer, single-consumer lock-free FIFO queues [94].
- Using the lock-free dequeue of Blumofe et al. [47] for the work-stealing scheduler.

The **first suggestion** would reduce the number of context switches caused by contention over channels, which would improve performance in two ways. First, overheads of context switching, scheduling and contention over run-queues would be reduced. Second, less obviously, avoiding a context switch also avoids a *stack switch*, which in turn reduces pressure on data caches.

The refined locking strategy involved in the first suggestion combines busy-waiting with spinning on the mutex. In this case, the mutex would be implemented as an integer whose value would reflect whether the mutex is taken, and if so, the *reason* for it being taken. There are only two such reasons:

- The channel's capacity is being enlarged.
- The process on the other end is accessing the channel.

In the first case, all messages in the channel's buffer must be copied to a new buffer. Copying is a heavy-weight operation, so the KP trying to lock the mutex would use busy-waiting. In the second case, the mutex will be held for a very short time if KPs communicate mainly with small messages. The KP trying to lock the mutex would then first spin on the mutex and yield if the mutex cannot be acquired after a preset number of iterations. The number of iterations is determined according to the *maximum* size $M$ of "small" messages: copying $M$ bytes or less should take less time than to yield to another KP. Since the low-level context switch accesses two different machine contexts, (see section 4.1.2.2), the *lower* bound on $M$ in our implementation is $M \geq 64$ bytes.[8]

The **second suggestion** would improve scheduler's performance by eliminating waiting on mutexes that protect runner's queues. However, using this data-structure would prevent us from implementing yielding, which is needed for busy-waiting mutexes, as well as the improvement of the work-stealing algorithm that we described in section 4.2.2.4.

We also deem that high contention over run-queues generally means that there is not enough work in the whole application. This problem is better solved by reducing the number of runners to match the application's average parallelism level, or by using fine-grained parallelism, thus increasing the number of KPs in the network and reducing the probability of a runner finding its queue empty.

[8]Figure 4.2 suggests that 240 bytes are used for registers. First, the structure is overdimensioned and padded to fit the cache line boundary. Second, only 8 64-bit registers *must* be saved and restored during context switch on AMD64.

## 5.3. Evaluation of load-balancing methods

In the first part of this section, we investigate the relative merits of the work-stealing (WS) and graph-partitioning (GP) methods presented in section 4.2. As previously, our metrics of interest are real (wall-clock) running time and speedup. We also introduce a new metric that reflects the number of messages communicated between processes assigned to the same CPU or to different CPUs. This metric is interesting for a possible future distributed version of Nornir because communication between processes on different machines is orders of magnitude more expensive than within the same machine.

In the second part of this section, we investigate the effects of contention over run-queues on performance. We also evaluate our modified work-stealing algorithm, described in section 4.2.2.4, against the original algorithm.

### 5.3.1. Comparison of work-stealing and graph-partitioning

In this section, we present the results of our comparative evaluation of work-stealing (WS) and graph partitioning (GP) load-balancing algorithms described in section 4.2. Our motive for this study is to investigate whether the GP method can perform better than WS on a multi-core machine, and if so, on which workloads. The GP algorithm can potentially improve performance in two ways:

- By reducing migration of KPs between runners, thus also reducing contention over run-queues.
- By reducing communication between KPs assigned to different runners, thus also reducing contention over channels.

In our evaluation, we have used the ring, H.264, RND-A and RND-B benchmarks described in section 5.2.1, as well as the MapReduce implementation of the k-means algorithm described in section 3.3.3.

#### 5.3.1.1. Methodology

Nornir has been configured to use the m:n work-stealing scheduler with deadlock detection *disabled*, detailed accounting *enabled*, and default channel capacity of 64 messages. In order to obtain more general results, we have disabled deadlock detection and checked that the benchmark did not terminate early due to unresolved artificial deadlock. The benchmarks have been compiled and run on the same hardware and OS as described in section 5.1.1. We have also taken care to account for NUMA issues, as we have described in section 5.2.3.1. Our discussions are based on *median* of 9 measurements. The reasons for choosing median instead of mean are explained below.

**Metrics** In addition to real running time and speedup, we introduce two additional metrics, *local traffic ratio* and *message throughput*, that we define with the help of traffic

volume. We define the *local traffic volume* as the as the number of messages exchanged between processes assigned to the same CPU at the time message was sent; remote traffic volume is defined analogously. The local traffic ratio $\alpha$ and message throughput $\beta$ are calculated as

$$\alpha = \frac{l}{l+r} \qquad \beta = \frac{l+r}{t}$$

Here, $l$ and $r$ are local and remote traffic volume ($l + r$ is thus the total traffic volume), and $t$ is the total running time.

**Benchmark parameters** In the **ring** benchmark, we used 1000 processes and 1000 round-trips, totaling $10^6$ [send $\rightarrow$ context switch $\rightarrow$ receive] transactions.

For the **k-means** benchmarks, we set the number of processes in each stage to 128, and the workload consisted of 300000 randomly-generated integer points contained in the cube $[0, 1000)^3$ to be grouped into 120 clusters.

The **H.264** benchmark was configured to "encode" 30 frames at rate of 1 frame per second, with the number of workers in each parallelized stage varying over the set $\{128, 256, 512\}$.

For the **random KPN** benchmarks we have used the same graphs in the same setup as described in section 5.2.3.1.

In addition to the above parameters, the GP policy introduces the idle time parameter $\tau$, explained in section 4.2.3.2. We have varied $\tau$ over the set $\tau \in \{8, 16, 24, \ldots, 256\}$ for each combination of other parameters.

**Presentation of results** To simplify our presentations, we have computed the *best* value of $\tau$ for each combination of the other parameters. However, we cannot simply select the $\tau$ value whose result set, consisting of 9 measurements, has the lowest mean (or median) running time. The reason is that consecutive runs of an experiment with the *same* combination of $d$ and $\tau$ can have high variance.

Thus, to choose the best $\tau$ for a given combination of other parameters, we compare all result sets against each other and select the set $s(\tau)$ which compares smaller against the largest number of other result sets. If we find several candidate result sets, we choose the one corresponding to the smallest $\tau$. Formally, we choose the best $\tau$ as follows:

$$\tau = \min(\arg\max_{\tau \in T} |\{\tau' \in T : (\tau' \neq \tau) \wedge (s(\tau) < s(\tau'))\}|)$$

where $|X|$ denotes the cardinality of set $X$, $T = \{8, 16, \ldots, 256\}$, i.e., the set of $\tau$ values for which we have run the experiments, and $s(\tau)$ is the result set for the given $\tau$. To compare two result sets, we have used the one-sided Mann-Whitney U-test [95][9] with 95% confidence level; whenever the test for $s(\tau)$ against $s(\tau')$ reported a p-value less than

[9] Usually, the Student's t-test is used. However, it makes two assumptions, for which we do not know whether they are satisfied: 1) that the two samples come from a normal distribution 2) having the same variance.

| Experiment | $t_1$ | $t_2$ | $t_4$ | $t_6$ | $t_8$ |
|---|---|---|---|---|---|
| GP | 1.43 | 1.65 | 1.78 | 1.58 | 1.71 |
| WS | 1.33 | 2.97 | 2.59 | 2.66 | 2.86 |

Table 5.8.: Summary of ring benchmark results with 1000 workers on $10^6$ transactions. $t_n$ is *median* running time on $n$ CPUs.

0.05, we considered that the data set $s(\tau)$ comes from a distribution with stohastically smaller [95] running time.

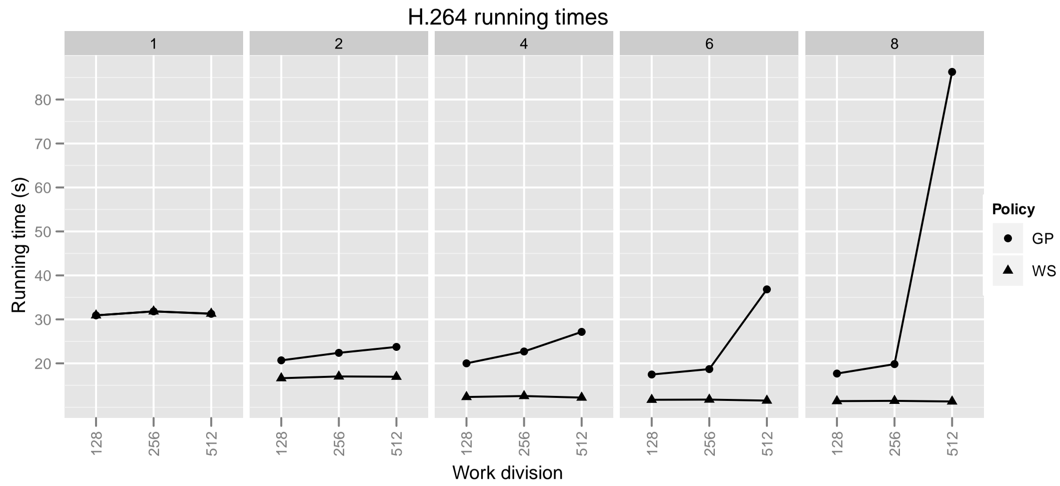
Once we have found the best $\tau$ for a given combination of other parameters, we have to choose a single representative run for the given set. This run will be used to represent not only the running time, but also other variables corresponding to that run, such as traffic volume. In our case, we choose the run corresponding to the *median* running time. This is also the reason for performing 9 measurements because the median of an *even* number of points is defined as the average of the two middle values, so this would generate artificial values.

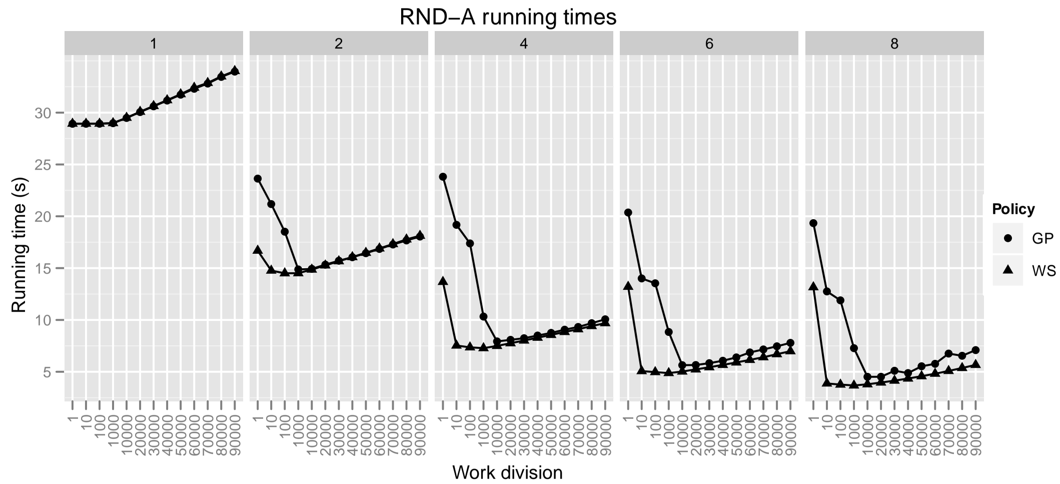
#### 5.3.1.2. Results

The **ring** benchmark measures scheduling and message-passing overheads of Nornir. Table 5.8 shows the results for 1000 processes and 1000 round-trips, totalling $10^6$ [send $\rightarrow$ context switch $\rightarrow$ receive] transactions. Under the GP policy, the median running time shows little variation with the number of CPUS. Under the WS policy it drastically increases when the number of CPUs increases from 1 to 2, after which it behaves irregularly, and does not exhibit any increasing or decreasing trend. This increase in running time is due to contention generated by additional CPUs trying to actively steal work when there is none. Similarly, the GP policy exhibits little variance with the number of CPUs because the CPUs are mostly accessing only their own run-queues. A CPU accesses another CPU's run-queue only to unblock the KP assigned to that CPU by the partitioning algorithm.

The **k-means** program, which executes on a MapReduce topology, is an example of an application that is hard to schedule with automatic graph partitioning. If the idle time parameter $\tau$ is too low, repartitioning runs overwhelmingly often, so the program runs *several minutes*, as opposed to 9.4 seconds under the WS method. When the $\tau$ is high enough, repartitioning runs only once, and the program finishes in 10.2 seconds. However, the transition between the two behaviours is discontinuous, i.e., as $\tau$ is gradually lowered, the behaviour abruptly changes from good to bad. Because of this, we will not consider this benchmark in further discussions.

Figure 5.20 shows median absolute running times for the **H.264**, **RND-A** and **RND-B** benchmarks. From the figure, it can be seen that the WS policy has *the least median running time for most workloads*; it is worse than the GP policy only on the ring benchmark (not shown in the figure; see table 5.8 instead) and the RND-B network when work division is $d \geq 30000$. At this point, performance of message-passing and scheduling



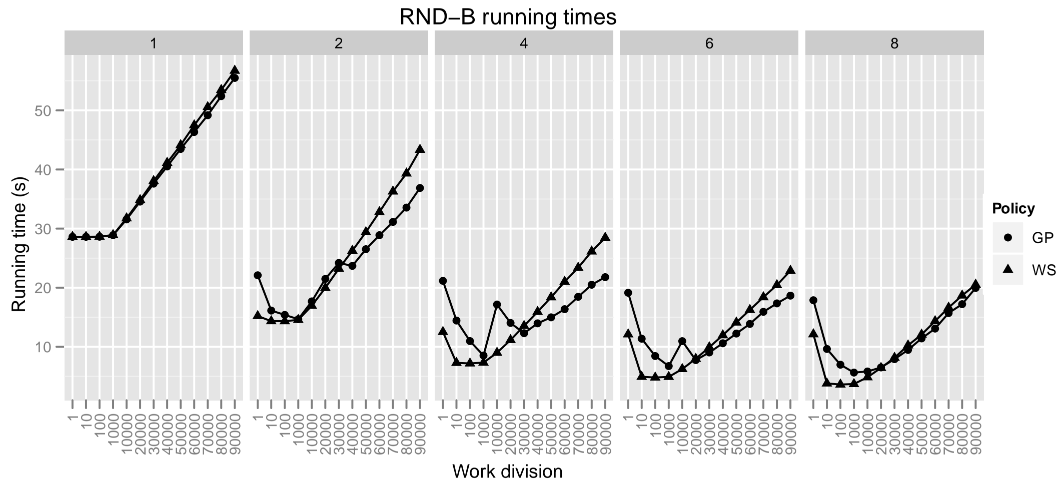




Figure 5.20.: Median running times for WS and GP policies on 1,2,4,6 and 8 CPUs.

becomes the limiting factor, so the running time increases proportionally with $d$. We can also see that that the GP policy shows severe performance degradation on the H.264 benchmark as the number of workers and the number of CPUs increases. This is caused by the limited parallelism available in the H.264 network; the largest speedup under WS policy is $\sim 2.8$. Thus, the runners accumulate idle time faster than load-balancing is able to catch-up, so repartitioning and process migration frequently takes place (90 – 410 times per second, depending on the number of CPUs and workers). The former is not only protected by a global lock, but its running time is also proportional with the number of partitions (CPUs) and the number of nodes in the graph, as can be clearly seen in the figure.

Figures 5.21, 5.22 and 5.23 show further data related to H.264, RND-A, RND-B benchmarks: speedup relative to 1 CPU, local traffic ratio and message throughput. WS achieves consistently better peak speedup than GP: at 512 workers for H.264, at $d = 1000$ for RND-A and $d = 100$ for RND-B. For the random KPNs, the speedup is linear in the number of CPUs, and the speedup on H.264 is limited by data-dependencies in the network.

By cross-referencing the speedup and local traffic ratio figures for the same $d$, we can see that the GP speedup has no correlation with peaks and valleys of the local traffic ratio, which constitutes over 70% of all message traffic on any number of CPUs. The local traffic ratio decreases proportionally with the number of CPUs under WS, but it shows only slightly decreasing trend under GP.

However, we can correlate the WS peak speedup and the lower knee of the message throughput curve for both random graphs. For RND-B, which is the only benchmark where GP has a (slight) advantage over WS, the message throughput under GP becomes greater than throughput under the WS policy for $d \geq 30000$. This coincides with upper knee of the throughput curve and the point where speedup under GP speedup becomes greater than WS speedup.

Figure 5.24 shows the relation between the local volume ratio and the number of busy-wait loop iterations per second for RND-A and RND-B benchmarks; in the H.264 benchmark too few messages were exchanged for obtaining a meaningful result. Each line corresponds to an experiment for a single work division factor and passes through exactly two points. The left point corresponds to the lower ratio (WS policy), and the right point to the higher ratio (GP policy). Since the total spin count is incremented each time a KP encounters a locked mutex protecting a channel, the figure indicates that GP reduces contention over channels.

Figure 5.25 uses the box-and-whiskers plot[10] to show the distribution of running times and number of repartitionings achieved for all benchmarked $\tau$ values of GP policy. The plots show the running time and the number of repartitions for the **RND-B** benchmark on 8 CPUs and $d = 1000$. From the graphs, we can observe several facts:

[10]This is a standard way to show the distribution of a data set. The box's span is from the lower to the upper quartile, with the middle bar denoting the median. Whiskers extend from the box to the lowest and highest measurements that are not outliers. Points denote *outliers*, i.e., measurements that are more than 1.5 times the box's height below the lower or above the upper quartile.

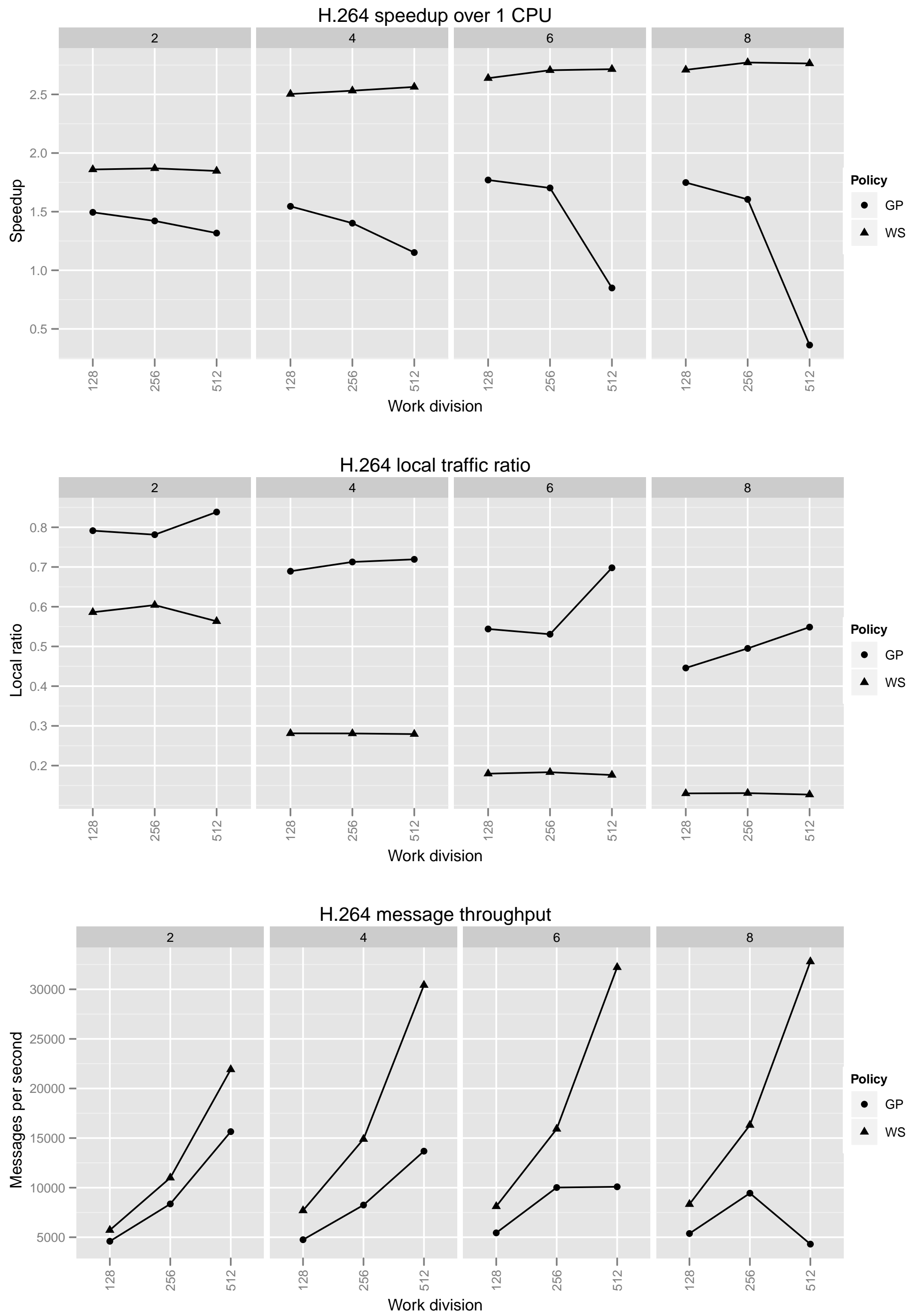
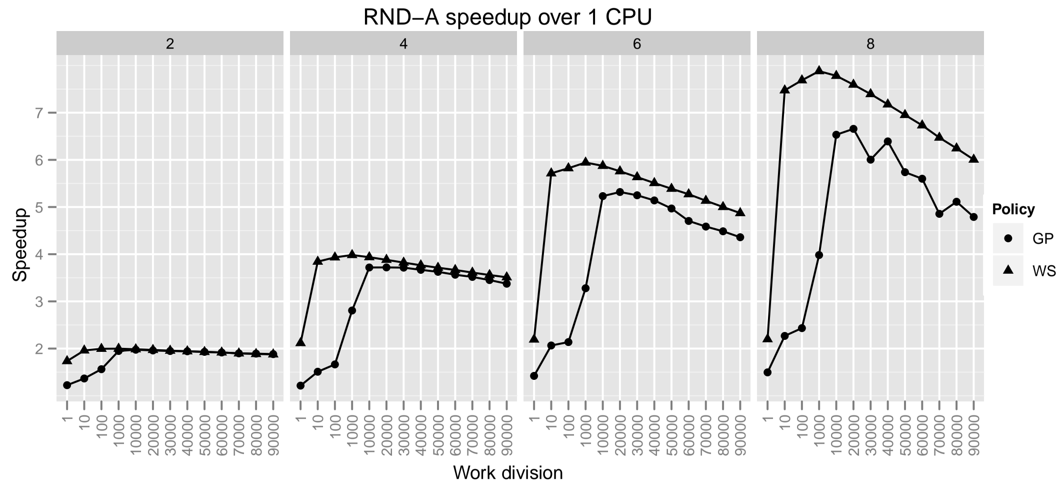
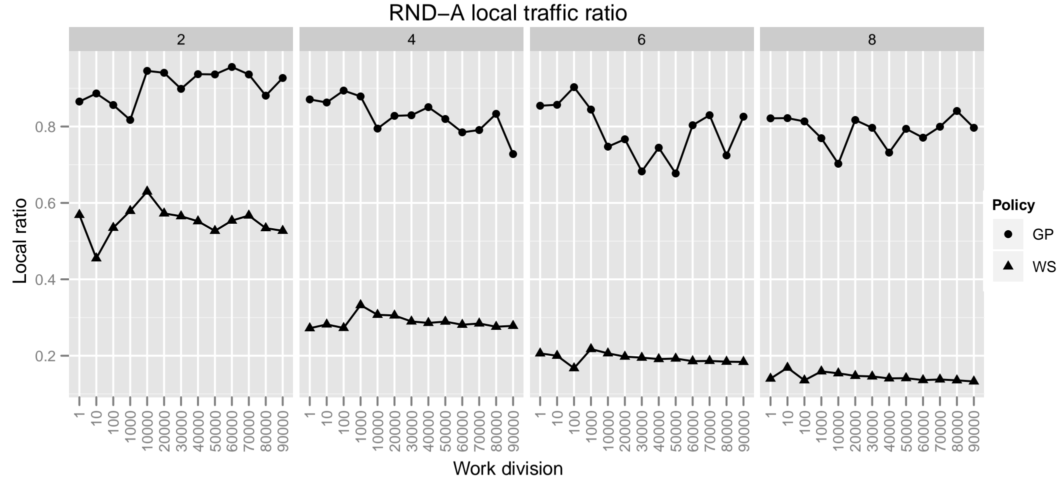



Figure 5.21.: H.264 speedup, local traffic ratio and message throughput.



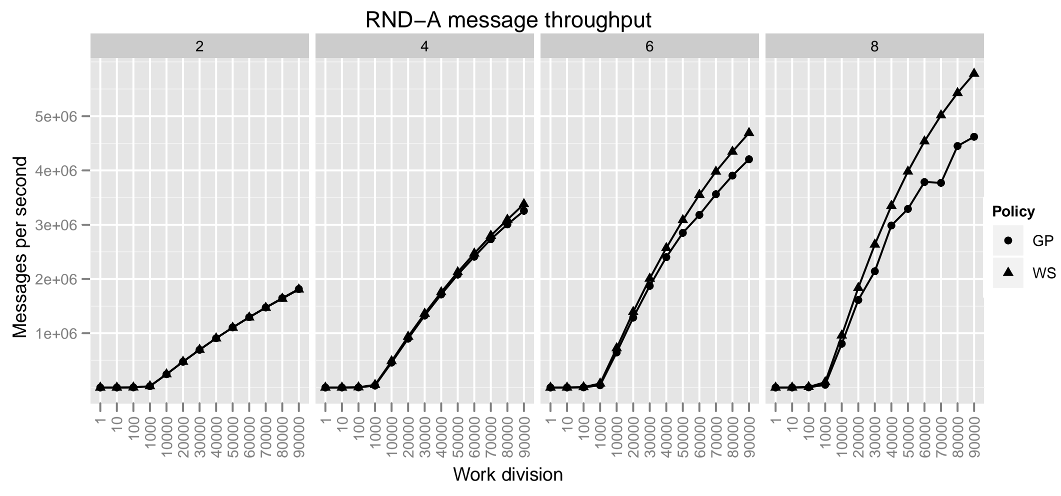
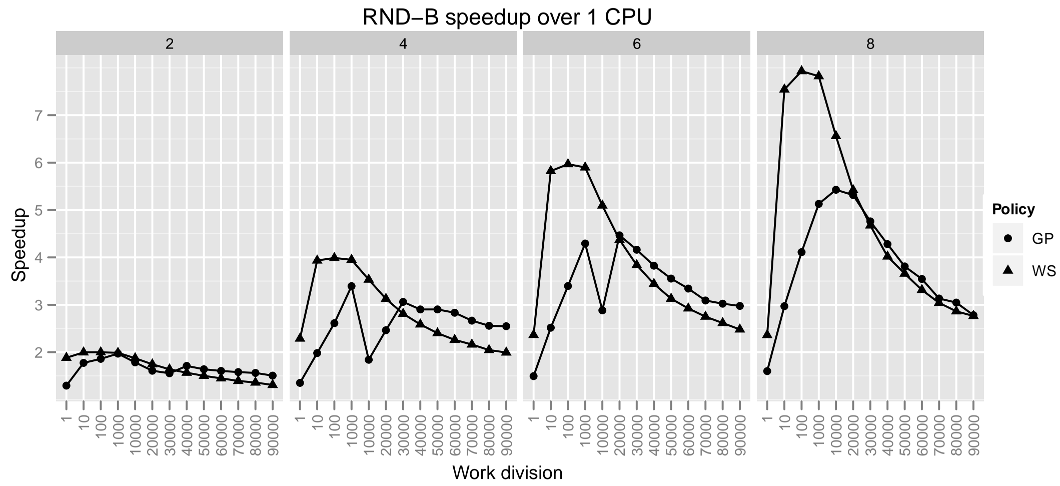
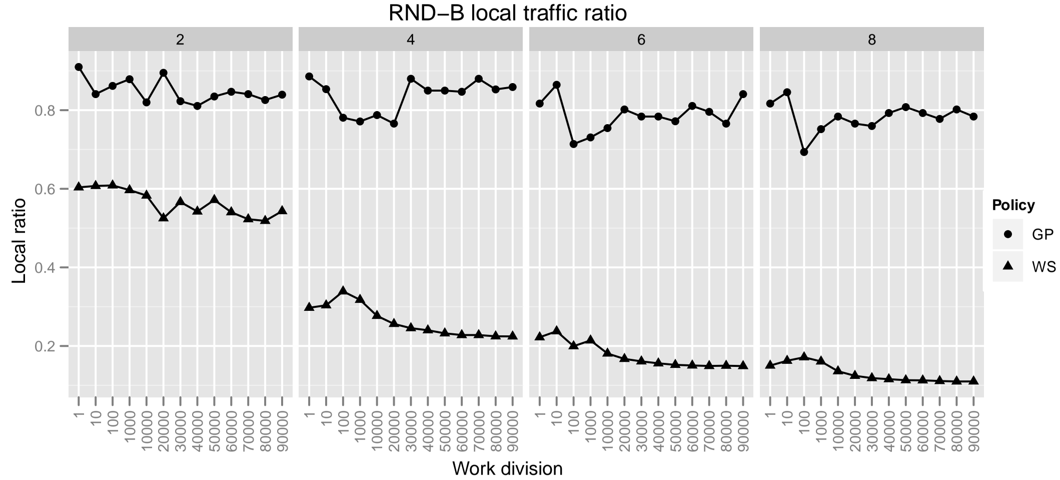




Figure 5.22.: RND-A speedup, local traffic ratio and message throughput.



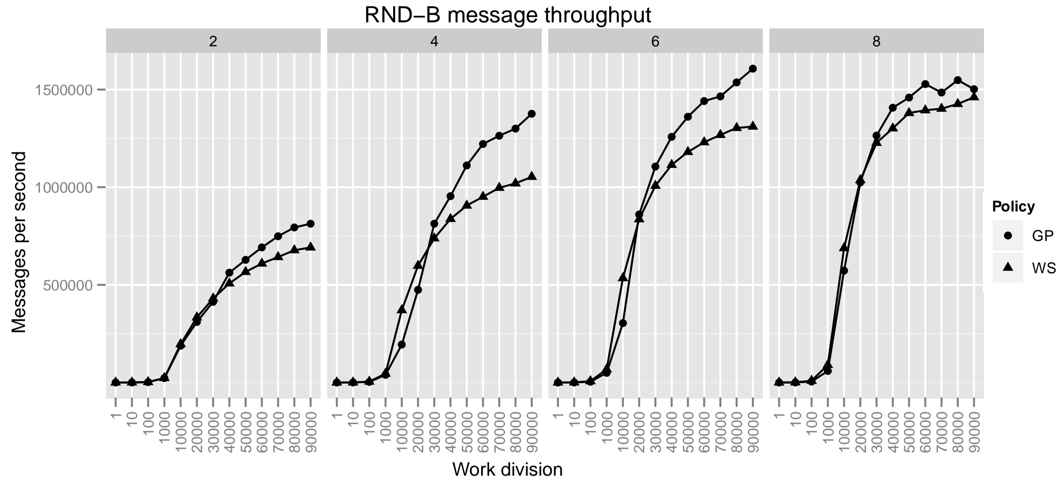




Figure 5.23.: RND-B speedup, local traffic ratio and message throughput.

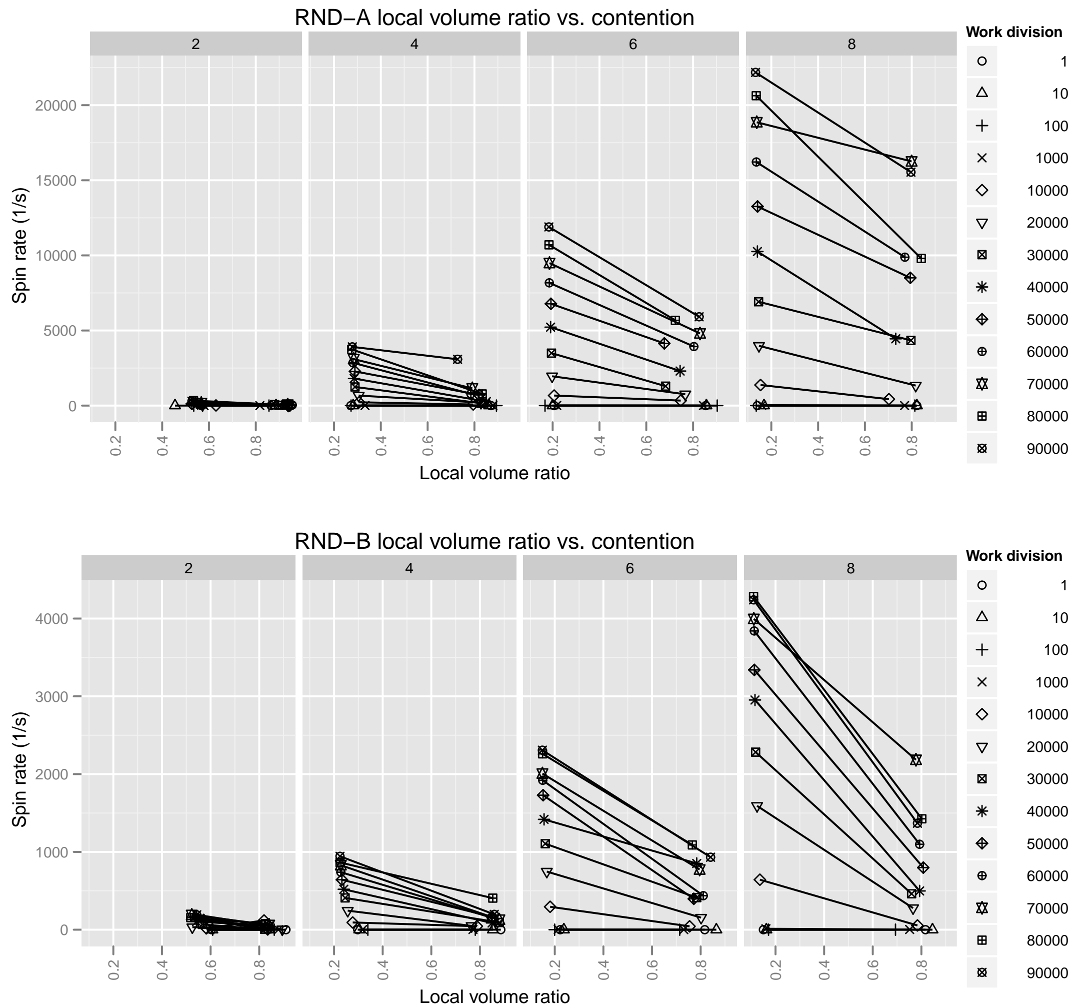



Figure 5.24.: Correlation between the local volume ratio and contention on channels on 2, 4, 6 and 8 CPUs.

- WS has negligible variation in running time (the single line at $\tau = 0$) in comparison with GP.
- The number of repartitionings is inversely-proportional with $\tau$, but no clear correlation with either variance or median running times is apparent.
- When $\tau \geq 72$, the *minimal* running times under the GP policy show rather small variation.

We have also investigated the relative performance of GP and WS policies, but now considering the *minimal* real running amongst all GP experiments for all values of $\tau$.

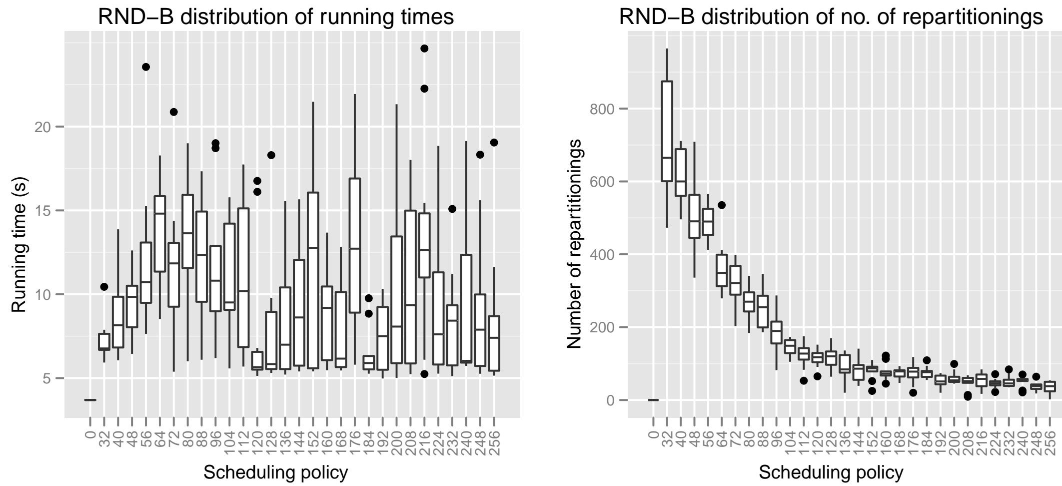



Figure 5.25.: Illustration of GP variance in running time for RND-B on 8 CPUs and $d = 1000$, which is one step past the WS peak speedup. x-axis is the value of the idle time parameter $\tau$ for the GP policy. $\tau = 0$ shows the results for WS.

The plots (not shown) are very similar to those of figures 5.22 and 5.23, except that GP speedup is slightly (< 2% on 8 CPUs) larger.

#### 5.3.1.3. Discussion

We have analyzed performance of graph-partitioning and work-stealing load-balancing methods for executing various workloads. Our findings can be summarized as follows:

- WS gives best performance, with speedup proportional with the number of CPUs, *provided* that there is enough parallelism in the network.
- GP and WS show similar patterns in running time, but GP never achieves the same peak speedup as WS.
- There exists an optimal work division $d$ at which the largest speedup is achieved; this value is different for WS and GP.
- Increasing $d$ beyond peak speedup leads to a sharp increase in message throughput. This increase quickly degrades performance because message-passing and context switch overheads dominate the running time.
- GP has large variance in running time; neither the median running time nor its variance is correlated with the idle time parameter $\tau$.
- GP achieves a greater proportion of local traffic than WS, and this ratio decreases very slightly as the number of CPUs increases. The proportion of local traffic under WS decreases as the number of CPUs increases.

- We have not found any apparent correlation between message locality, which also reduces contention over channels, and running time or speedup.

Regarding GP, there are two main obstacles that must be overcome before it can be considered as a viable approach for scheduling general-purpose workloads on multiprocessor machines:

- Unpredictable performance, as demonstrated by figure 5.25.
- The difficulty of automatically choosing the right moment for rebalancing.

As our experiments have shown, monitoring the accumulated idle time over all CPUs and triggering rebalancing after the idle time has exceeded a threshold is not a good strategy. Thus, work-stealing should be the algorithm of choice for scheduling general-purpose workloads. Graph partitioning should be reserved for specialized applications using fine-grained parallelism, that in addition have enough knowledge about their workload patterns so that they can "manually" trigger rebalancing.

### 5.3.2. Limits of work stealing

The previous evaluation has shown that work-stealing is superior to graph partitioning scheduling algorithm on multi-core machines. In this section, we investigate the effects of contention over run-queues on performance of the work-stealing algorithm. We have evaluated both the original work-stealing algorithm, which we shall denote as WS-CUR, and our modified work-stealing algorithm, described in section 4.2.2.4, which we shall denote as WS-LAST.[11]

For this purpose, we have also designed a new synthetic benchmark, scatter/gather, which will be central in our evaluations. This benchmark represents a pattern common for embarrassingly parallel workloads, such as AES in ECB or CTR mode. The scatter/gather topology also emerges when several MapReduce instances are executed such that the result of the previous MapReduce operation is fed as the input to the next, as was the case in the k-means application described in section 3.3.3.

#### 5.3.2.1. Methodology

We have configured Nornir to use m:n work-stealing scheduler with deadlock detection *disabled*, detailed accounting *enabled*, and default channel capacity of 64 messages. The benchmarks have been compiled and run on the same hardware and OS as described in section 5.1.1. The experiments have been run on 1 CPU, to obtain reference values, and on 8 CPUs. We have chosen to skip the intermediate values because we are most interested in performance characteristics of work-stealing on more CPUs. As previously, we have also taken into account NUMA effects, and our discussions are based on *median* of 9 measurements.

[11] These acronyms are also meant to be mnemonical: WS-CUR unblocks a KP to the *current* runner, which is also the original ("current") work-stealing algorithm. WS-LAST unblocks a KP to the *last* runner it was running on before it blocked.

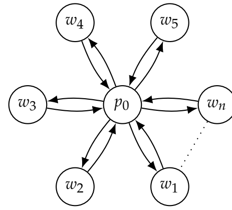



Figure 5.26.: Worker processes connected in a scatter/gather topology to a central process.

**Metrics** As previously, our most important performance indicators are the real running time and speedup. We also introduce a new metric, *steal rate*, which we define as the ratio $n_i/t$. Here, $n_i$ is the total number of steal attempts (both successful and unsuccessful) performed by all runners during program execution, and $t$ is the program's running (real) time.

**Scatter/gather benchmark** The **scatter/gather** network has a single central process ($p_0$) connected to $n$ worker processes (see figure 5.26). $p_0$ executes *m rounds* of the following procedure. First, it sends to each worker a single message representing $w = T/d$ CPU seconds, where $d$ is the work division factor. Upon receipt of a message, a worker calls the `burn_cycles` function, shown in figure 5.7, with argument $wT_0$, to consume the specified amount of CPU time, and then it sends a reply message back to $p_0$; the $T_0$ constant has been explained in section 5.2.1.4. In the meantime, $p_0$ waits to receive a reply from all workers, and then begins the next round of distributing work to workers, thus acting as a barrier [39]. The total workload executed by the workers is $W = nmw = nmT/d$ CPU seconds.

#### 5.3.2.2. Results

Our presentation of results is presented as a series of five experiments since each experiment opened new questions that had to be explained. In the first experiment, we have run all three benchmark programs (ring, pipeline and scatter/gather) under both WS-CUR and WS-LAST policies. In the second and third experiment, we have run only the scatter/gather benchmark because it was the only one from the first experiment that exhibited behavior warranting further study. These experiments have revealed that WS-CUR and WS-LAST exhibit the same qualitative behavior, so we have run the last two experiments (fourth and fifth) using only the scatter/gather benchmark and only WS-LAST algorithm.

**Experiment 1** In this experiment, we have first run the **ring** benchmark with 50 and 1000 processes, and the number of round-trips has been adjusted so that the total number of transactions is $10^6$. The running times under of both configurations on 1, 2, 4, 6 and 8 CPUs under both scheduling algorithms are shown in table 5.9. From the table, we can see that the running time with 1000 processes on one CPU under the same algorithm is higher than with 50 processes. Since both scheduling algorithms use constant time

| Experiment | $t_1$ | $t_2$ | $t_4$ | $t_6$ | $t_8$ |
|---|---|---|---|---|---|
| WS-LAST/50 | 0.87 | 3.00 | 4.53 | 6.42 | 8.44 |
| WS-CUR/50 | 0.92 | 2.93 | 2.53 | 2.65 | 2.83 |
| WS-LAST/1000 | 1.34 | 3.03 | 4.49 | 6.42 | 8.42 |
| WS-CUR/1000 | 1.33 | 2.97 | 2.59 | 2.66 | 2.86 |

Table 5.9.: Summary of ring benchmark results with 50 and 1000 workers on $10^6$ transactions. $t_n$ is *median* running time on $n$ CPUs.

to select a runnable process, and since both configurations perform the exact same number of context switches, we believe that the increase is caused by lower-level issues such as increased cache miss rate. At two or more CPUs, contention over run-queues shadows these effects, so there is little difference between running times with 50 and 1000 processes under the same policy. On the other hand, there is a big qualitative difference between WS-CUR and WS-LAST policies as the number of CPUs increases beyond two. The running time of the benchmark under WS-CUR is relatively stable, whereas it grows proportionally with the number of CPUs under the WS-LAST policy.

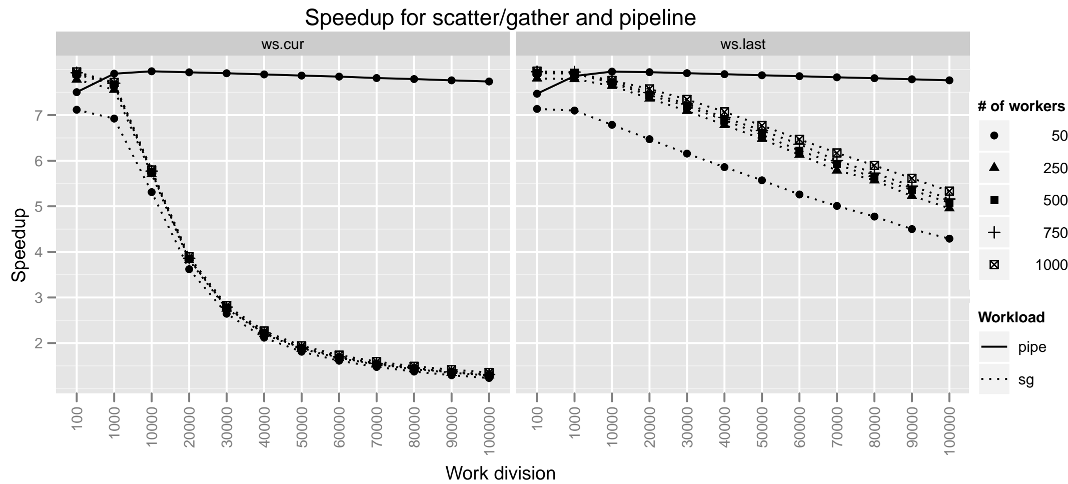
For the **scatter/gather** and 50-stage **pipeline**, we used $T = 1$ and we varied the work division over the set $d \in \{100, 1000, 10000, 20000, \ldots, 100000\}$, i.e., each message carries a workload of $1/d$ CPU seconds. Additionally, in the scatter/gather benchmark, we varied the number of workers over the set $n \in \{50, 250, 500, 750, 1000\}$, and we computed the number of rounds as $m = \lfloor 50d/n \rfloor$. In this way, the total amount of work distributed by the central process was held constant at $W = 50$ CPU seconds.

Figure 5.27 shows speedup and steal rate for pipeline and scatter/gather benchmarks.

From the figure, we can observe that the **pipeline** benchmark has almost the same performance characteristics, regardless of the scheduling strategy algorithm. The slight increase in speedup as $d$ reaches 10000 is due to better pipeline utilization, as has already been explained in section 5.2.3.2.

The **scatter/gather** benchmark exhibits drastically different performance characteristics: its performance starts degrading much more rapidly at $d = 10000$ under the WS-CUR algorithm than under the WS-LAST algorithm. At $d = 10^5$, the speedup is barely greater than 1 under the WS-CUR algorithm, but it is still above 4 under the WS-LAST algorithm. We can also see that the sharp drop in speedup for the WS-CUR algorithm coincides with the sharp rise in steal rate.

At $d = 30000$, the steal rate reaches saturation under the WS-CUR algorithm, but the speedup continues to fall because each runner has to wait longer time before it gains access to its own or another runner's run-queue. At $d = 30000$, the waiting time is $\sim 4.5\mu s$, while at $d = 10^5$, the waiting time is $\sim 5.75\mu s$. This prolongation is caused by two simultaneously interacting factors. First, $p_0$ unblocks all workers, which the WS-CUR algorithm places on the *same* runner ($R_0$) where $p_0$ is executing. When $p_0$ has blocked, all workers are in the run-queue belonging to $R_0$, while run-queues of the other runners are empty. Therefore, the other runners immediately begin stealing, and each



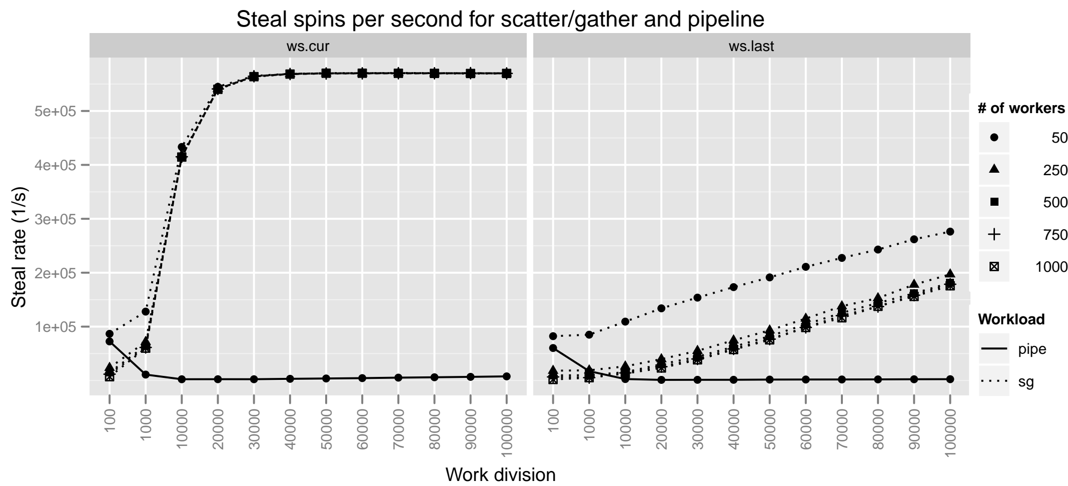




Figure 5.27.: Scatter/gather and pipeline on 8 CPUs: speedup and steal rate vs. work division.

must perform $7/2 = 3.5$ steal attempts on average[12] before it "hits" the (only) runner having a non-empty run-queue, $R_0$.

Second, as $d$ increases, the useful work performed by each worker decreases. Since each runner steals only a single process, they will engage in stealing more often. This causes in turn greater contention over the run-queue belonging to $R_0$, which delays both $R_0$ and other runners in dequeueing or stealing worker processes. As long as $p_0$ has more work distribute, contention also delays $R_0$ in *enqueueing* the workers.

We can also see that speedup of the scatter/gateher benchmarks increases with the

[12] $(N-1)/2$ in the general case, $N$ being the number of CPUs. $N-1$ is used because a runner never attempts to "steal" work from itself.

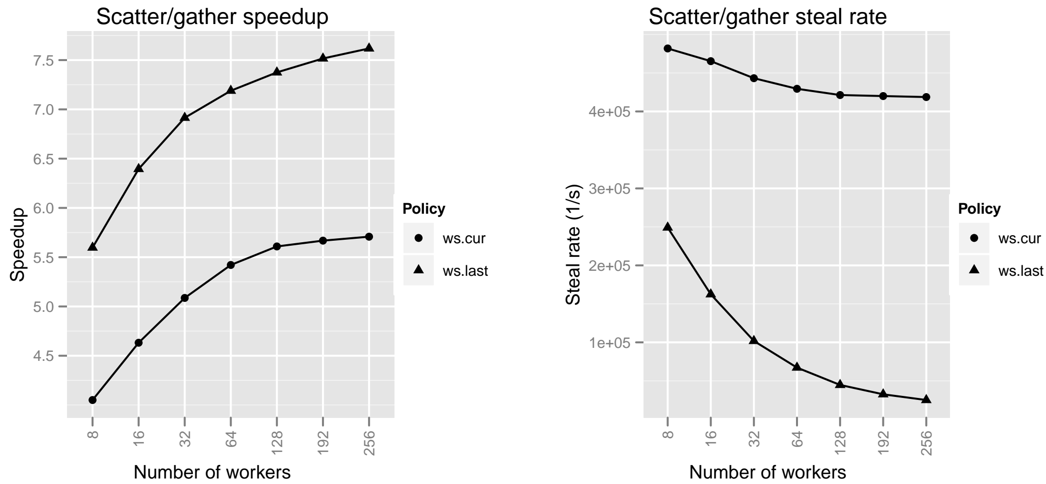



Figure 5.28.: Scatter/gather on 8 CPUs: speedup and steal rate vs. number of workers.

number of workers, independently of work division and the work-stealing variant. The largest increase in speedup is at the transition from 50 to 250 workers, and it coincides with the drop in steal rate. Since 50 workers is already a much larger number than the number of CPUs used in the experiment (8), we have designed another experiment to find the causes of this behavior.

**Experiment 2** In this experiment, we further investigate the relation between the number of workers and speedup, which is clearly visible in figure 5.27. We have fixed work division to $d = 10000$, which corresponds to $\sim 100\mu$s of processing time per message, the number of rounds to $m = 12500$, and we have varied the number of workers over the set $n \in \{8, 16, 32, 64, 128, 192, 256\}$. We have chosen these numbers based on the results shown in figure 5.27. Work division $d = 10000$ is the critical value at which speedup starts decreasing rapidly, yet it is still reasonably high. The number of rounds $m$ is adjusted so that the total amount of work performed by all workers for $n = 8$ is at least 10 seconds. In general, the total workload in this benchmark can be calculated as $W = n \cdot 12500 \cdot T/10^4 = 1.25n$ CPU seconds, which also sets the lower bound for running time on 1 CPU. Having more workers or rounds would lead to unreasonably long running times because the number of rounds and work division are held constant. This is unlike the previous experiment, where the number of rounds $m$ was adjusted in order to hold $W$ constant.

In figure 5.28 we can see that speedup increases with the number of workers $n$; at $n = 256$, WS-CUR has a speedup of 5.7, and WS-LAST has speedup of 7.6. The general trend is that the *rate* of the increase diminshes with $n$, and that increasing $n$ further would not significantly affect speedup. Furthermore, as $n$ increases, the difference in speedup between the two algorithms also increases. We can also see that WS-CUR has much larger steal rate than WS-LAST, and that the difference increases with $n$.

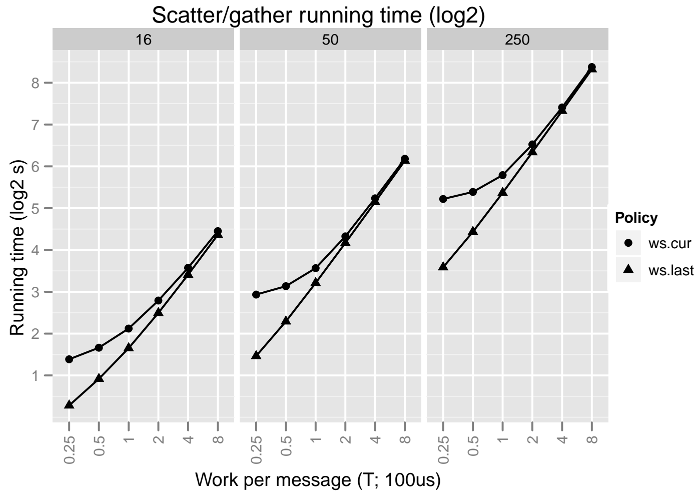



Figure 5.29.: Scatter/gather on 8 CPUs: $\log_2$ of running time vs. work per message.

From these results we conclude that increasing the number of workers $n$, which also increases the total work amount $W$, indeed does reduce contention over run-queues and improves performance. However, the performance improvement with increasing $n$ is limited and did not meet our expectations. We thus conjecture that the key factor in determining efficiency of WS on the scatter/gather benchmark is CPU time spent on processing a single message. To validate this assumption, we have designed another experiment.

**Experiment 3** In this experiment, we investigate the relation between work per message and speedup. We have fixed the number of rounds to $m = 12500$, varied the number of workers over the set $\{16, 50, 250\}$ and work per message over the set $w \in \{25, 50, 100, 200, 400, 800\}$ microseconds. In terms of the scatter/gather benchmark description, we have fixed work division to $d = 10000$ and varied $T$, which was fixed to 1 in the previous experiments, over the set $T \in \{1/4, 1/2, 1, 2, 4, 8\}$.

We have first noticed that the total running time grows *slower than linearly* with each doubling of work per message $w$ until a certain threshold (see figure 5.29). The effect is most visible at 16 workers, where we can observe that for $T < 4$, doubling its value leads to an increase on y-axis which is less than 1. Since the y-axis shows the base-2 logarithm of the running time, this means that a two-fold increase in $w$ leads to a *less than two-fold* increase in the running time. This is because CPUs start spending more time executing useful work than spinning in the stealing loop. Under WS-CUR, the effect is visible for all worker counts, while under WS-LAST, the effect is less pronounced at 50 workers and almost non-existent at 250 workers.

Figure 5.30 shows the relation between speedup and CPU time per message. We can see that both algorithms exhibit a significant performance improvement as T increases from $25\mu$s to $100\mu$s, and this increase in performance is correlated with the drop in

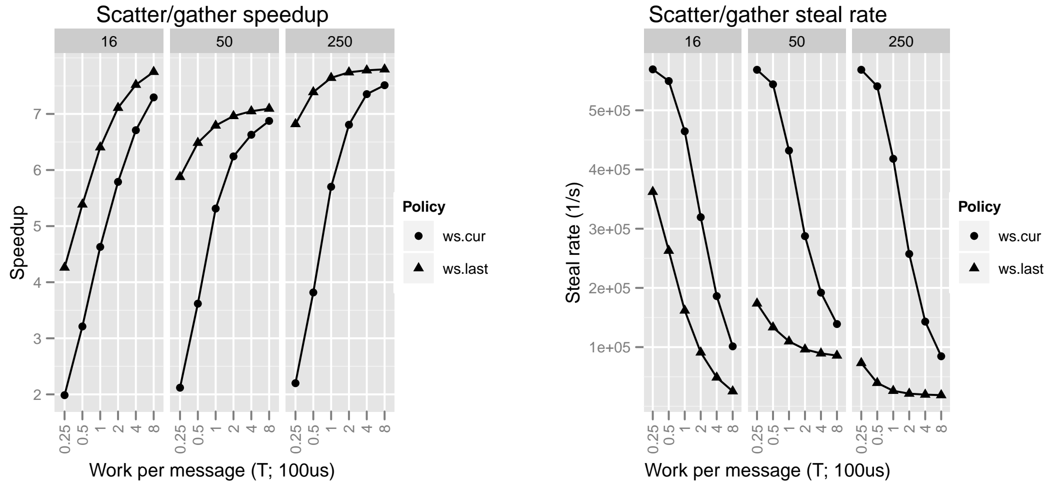



Figure 5.30.: Scatter/gather on 8 CPUs with 16 workers: speedup and steal rate vs. work per message.

steal rate. Performance continues to increase as $w$ grows towards 800$\mu$s per message, but much more so for the WS-CUR algorithm than for the WS-LAST algorithm. In other words, WS-LAST approaches the maximum speedup at lower values of $w$ which indicates that it is better suited for executing finely-grained parallel applications.

In this experiment, we have confirmed the conjecture from the previous experiment, namely that the efficiency of work-stealing is dependent on processing time per message. However, another anomaly has appeared: for $w = 800\mu$s, the speedup at 50 workers is less than the speedup at 16 and 250 workers. This anomaly is the subject of the next experiment.

**Experiment 4** To investigate how the number of workers affects speedup, we have fixed work division to $d = 10000$, work per message to $w = 800\mu$s, and varied the number of workers from 16 to 48 i steps of 1. Figure 5.31 shows the real (wall-clock) running time and speedup of WS-LAST on 8 CPUs.

The plots exhibit a distinct pattern: the running time increases step-wise with the number of workers $n$, with jumps occurring when $n \bmod 8 = 1$. Likewise, the speedup is worst when $n \bmod 8 = 1$ and best when $n \bmod 8 = 0$. The plots indicate that the workers execute in *batches*. In each round, $\lfloor n/8 \rfloor$ batches are executed in which all 8 CPUs are busy executing workers. If $n$ is not evenly divisible by 8, there will be an *overflow batch* in which only $n \bmod 8$ CPUs are busy executing workers, while the other CPUs are attempting to steal work from other CPUs. However, no additional work exists because the central process ($p_0$ in figure 5.26) starts a new round only after all workers have processed their messages in the current round. Thus, the overflow batch, if it occurs, stalls $8 - n \bmod 8$ CPUs, which decreases the total speedup in proportion with the number of the stalled CPUs.

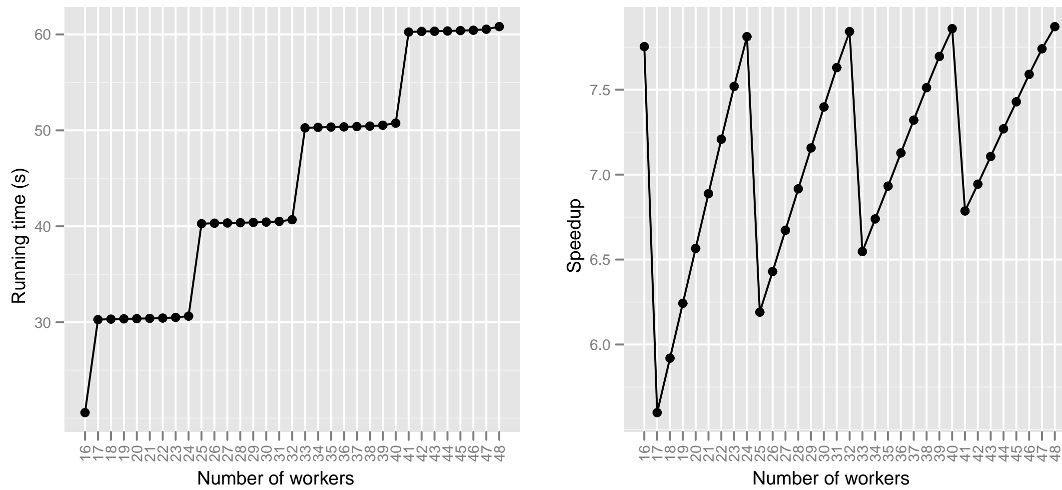



Figure 5.31.: Scatter/gather on 8 CPUs with 16–48 workers and $T = 8$: running time and speedup under WS-LAST.

The above reasoning also implies that the workers are running in synchrony. If it were otherwise, some CPU would finish executing its workers earlier, which would allow it to steal workers from other CPUs and reduce the size of the overflow batch. Stalling caused by the overflow batch would be reduced, if not eliminated, so the speedup curve would not exhibit the sawtooth shape with period 8 (the number of CPUs), as it does in figure 5.31. Thus, neither the delay caused by $p_0$ while distributing work, nor contention over run-queues are enough to disturb running in almost perfect lock-step. This in turn implies that the total running time in each round used by $p_0$ and the time spent on stealing is negligible compared with work amount per message (800$\mu$s), i.e., both are very efficient.

In figure 5.31, we can also see that speedup maximums, achieved for $n \bmod 8 = 0$, are relatively constant, while the speedup minimums, achieved for $n \bmod 8 = 1$, *increase* with $n$. This is because the fraction of the total work $W$ performed just by the overflow batch decreases as $n$ increases, which also decreases the time that the stalled CPUs spend in waiting. Thus, the ratio of waiting time and the total running time decreases, which leads to greater speedup.

**Experiment 5** To investigate how amount of work per message affects speedup and contention, we have fixed work division to $d = 10000$ and varied the number of workers from 16 to 24 (corresponding to one period of figure 5.31) and $w$ from 25 to 1600 $\mu$s per message. We have not run the experiment with a larger number of workers because figure 5.31 shows that speedup minimums become nearly 7 already at $n = 41$.

Figure 5.32 shows that we do not need to be particularly careful with choosing the number of workers when $T < 1$ because the difference in speedup is small. When $T \geq 1$, it becomes more important to correctly choose the number of workers in order to avoid

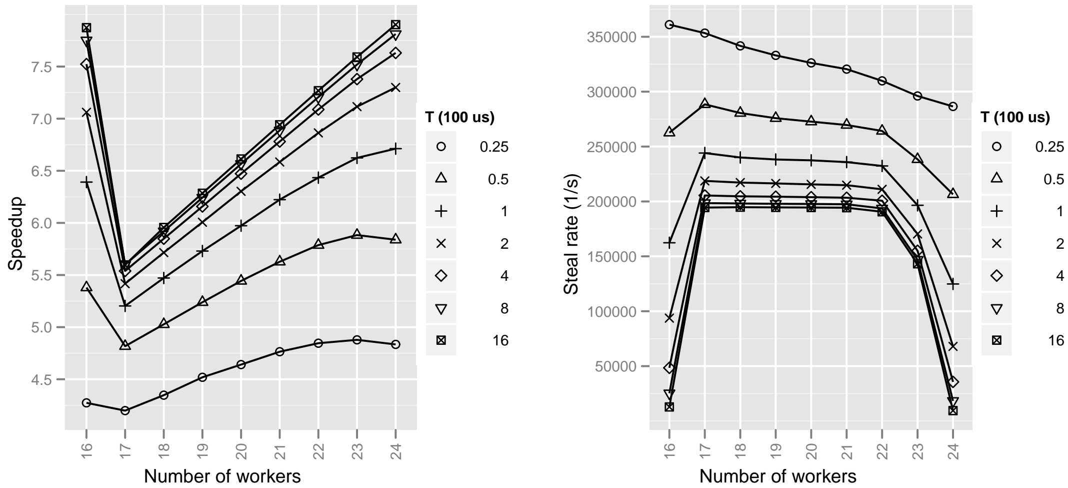



Figure 5.32.: Scatter/gather on 8 CPUs with 16–24 workers and varying work per message $T$: speedup and spin rate under WS-LAST.

the effects of the overflow batch described in the previous experiment. At $T = 1$, the speedup for $n = 24$ is 1.28 times grater than for $n = 17$; at $T = 16$, this factor becomes 1.39. Similarly, if $T$ is held constant, increasing the number of workers can avoid the performance loss caused by the overflow batch. In other words, the amount of work per message and the number of workers should not be chosen independently if one aims to achieve good performance. Larger work amounts per message give a better speedup, but they are also more sensitive to the number of workers.

To confirm the hypothesis about workers running in synchrony which we made in the previous experiment, we have modified the scatter/gather benchmark to perturb the work amount $T$ for each received message, thus obtaining the perturbed value $T'$. $T'$ is obtained by $T' = \alpha T$, where $\alpha$ is a random number drawn from a uniform distribution covering the interval $\alpha \in [1 - \iota/2, 1 + \iota/2]$. Here, $\iota$ is the jitter amount; e.g., jitter of 20% corresponds to $\iota = 0.2$. By drawing $\alpha$ from a symmetric interval centered at 1, we achieve that the *expected* value of $T'$ equals $T$, so the total expected amount of CPU time spent by workers will be the same as without perturbation.

Figure 5.33 show the effects of perturbing the work amount per message. We can see that perturbation indeed does help to smooth out the sawtooth shape that was so manifest in figure 5.32. It is, however, somewhat surprising that a relatively large perturbation of 30% in each direction, i.e., $\pm 60\mu$s peak-to-peak, is needed to smooth out the sawtooth shape. We can also see that perturbation can have either a positive or negative effect on speedup. When $n$ belongs to the lowest 1/4 of the sawtooth period ($n \in \{17, 18\}$), a more uneven distribution of work across messages (larger $\iota$) gives better speedup than a more even distribution. Conversely, when $n$ belongs to the upper half-period of the sawtooth, larger $\iota$ gives worse speedup. $n = 19$ is the cross-over point where speedup is independent of the jitter intensity. Thus, contrary to the

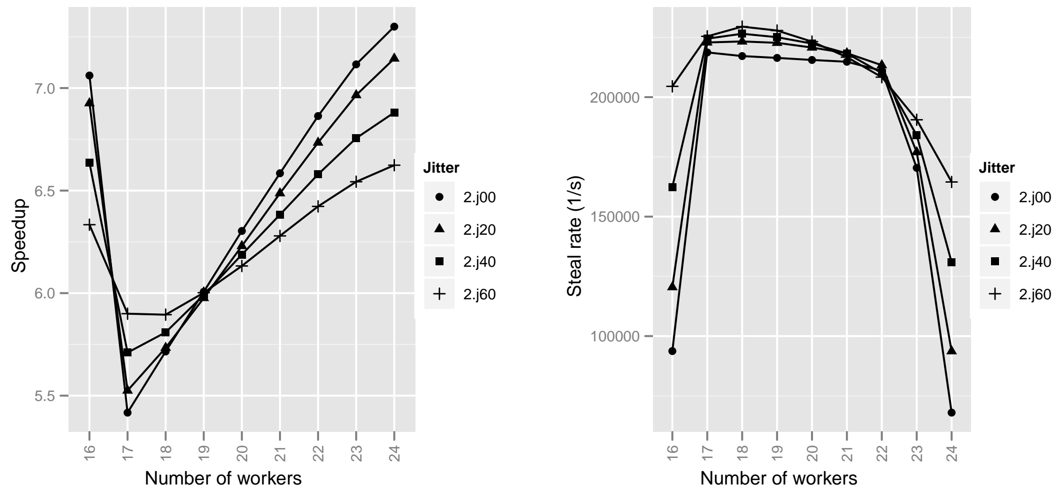



Figure 5.33.: Scatter/gather on 8 CPUs with 16–24 workers with $T = 2$ and jitter of 0, 20, 40 and 60%: speedup and spin rate under WS-LAST.

"common wisdom" of evenly distributing work, this experiment shows that somewhat *uneven* distribution of work might in certain cases achieve *better* speedup than even distribution.

The results of this experiment are also significant in a wider context. For example, common distributed MapReduce implementations [8, 27] will wait that all Map tasks finish before starting Reduce tasks [96]. Similarly, when several MapReduce are chained together, which is a common practice [9], the Map tasks of the next computation must wait that all Reduce tasks of the current computation finish. In other words, there is a barrier between each computation stage, which resembles our scatter/gather benchmark. The resemblance does not stop there, because the tasks run for approximately the same duration, i.e., they execute in "waves" [96], which are analogous to our batches. Thus we also recommend that MapReduce computations be configured so that the total number of workers in each of Map and Reduce stages is divisible by the total number of CPUs across all nodes involved in the computation.

#### 5.3.2.3. Discussion

We have evaluated performance of the original work-stealing algorithm (WS-CUR) and our modification (WS-LAST, described in section 4.2.2.4). The main goal was to understand the performance of WS under different workloads and different parallelism granularities. In this respect, our evaluation complements the work of Saha et al. [83]. Even though they emphasize that fine-grained parallelism is important in large-scale CMP design, they have not attempted to *quantify* parallelism granularity. By using a cycle-accurate simulator, they investigated the scalability of work-stealing on a CPU having up to 32 cores, where each core executes 4 hardware threads round-robin. Their

main finding is that contention over run-queues generated by WS can limit, or even worsen, application performance as new cores are added. They have found that applications perform better with static partitioning of load in such cases, but they have not described the algorithm used for static partitioning.

We deem that their results do not give a full picture about WS performance, because contention depends on three additional factors, neither of which is discussed in their paper, and all of which can be used to reduce contention:

- Decreasing the number of CPUs to match the *average parallelism* available in the application.
- Overdecomposing an application, i.e., increasing the total *number of processes* in the system proportionally with the number of CPUs.
- Increasing the *amount of work* a process performs before it blocks again.

The average parellelism is a property intrinsic to the problem at hand and little can be done to change it. Nevertheless, even if the number of CPUs is less than or equal than the application's average parallelism, the choice of other two factors can significantly influence the overall performance. Our evaluations thus focus on determining how to choose the number of processes and amount of work to achieve good scalability. In our evaluations, we have used the "rule of thumb" that an application has satisfactory performance on 8 CPUs if the speedup relative to the running time on 1 CPU is $\geq 7$.

The **ring** benchmark is anomalous in that only a single process is running at any moment, and that no work is performed per message. By executing this benchmark on 1 CPU, we have found that the median time of a single transaction is $\sim 0.9\mu$s with 50 processes, and $\sim 1.3\mu$s with 1000 processes. On several CPUs, there is a big qualitative difference in performance between WS-CUR and WS-LAST. The transaction time under WS-CUR jumps to $\sim 3\mu$s on 2 CPUs and remains approximately constant on up to 8 CPUs. On the other hand, the transaction time under WS-LAST increases linearly with the number of CPUs.

The 50-stage **pipeline** benchmark represents a common programming idiom. We have found that the best speedup (almost 8 on 8 CPUs) is achieved when all the channels in the pipeline are full, which minimizes the number of context switches. In this benchmark, we have divided the total work amount (50 CPU seconds) into chunks as small as $10\mu$s, with only small performance degradation. The best speedup of 7.95 was achieved was achieved for work division of $100\mu$s per message, which is also the smallest chunk size that fills all channels. With work amount of $10\mu$s per chunk the speedup decreased to 7.76.

The **scatter/gather** benchmark represents another common programming idiom. The performance of this pattern is hard to characterize, and we had to perform five different experiments to fully understand it. There is a big qualitative difference between WS-CUR and WS-LAST: as work per message $w$ falls below $1000\mu$s, both algorithms degrade in performance, but WS-CUR much more so than WS-LAST. As $w$ drops from $1000\mu$s to $100\mu$s with 50 workers, the drops from 6.9 to 5.3 under WS-CUR, but much less,

from 7.1 to 6.8 under WS-LAST. The trend continues: as $w$ grows, the speedup under WS-CUR appears to decay exponentially, but under WS-LAST it drops only linearly (see figure 5.27). The same figure shows that speedup depends on the number of workers $n$: there is a significant gain as $n$ increases from 50 to 250, and somewhat smaller gain as $n$ increases towards 1000.

Further experiments have shown that the best speedup is obtained with relatively large $w \geq 100\mu$s, but that in those cases, $n$ must be large or divisible by the number of CPUs $N$ ($N = 8$ in our case). When $n$ is small (in our case, 16 – 24), and *not* divisible by $N$, an uneven distribution of work may in some cases *improve* speedup. Also, WS-LAST reaches the speedup plateau at larger work divisions than WS-CUR, which indicates that WS-LAST is better-suited to fine-grained parallelism, at least with scatter/gather workloads.

As expected, we have also observed a consistent inversely-proportional relationship between speedup and the steal rate. On 8 CPUs, a satisfactory speedup ($\geq 7$) is obtained as long as all CPUs together perform fewer than $\sim 10^5$ steal attempts per second.

WS-CUR and WS-LAST have otherwise almost the same performance characteristics for other workloads of section 5.2.1. Based on the above discussions, we give several guidelines for using work-stealing scheduling algorithms on machines with up to 8 CPUs:

- WS gives good performance if each process performs at least $100\mu$s ($\sim 100$ times larger than transaction cost) of work, independently of the network topology, between it is waken up and it blocks again.

- Alternatively, the rate of steal attempts, summed over all CPUs, should be held under $10^5$ per second. Depending on the network topology, this allows in some cases even more fine-grained parallelism, down to $10\mu$s of work per message.

- An unexpected result is that speedup of the scatter/gather topology is heavily influenced by the number of worker KPs. We thus recommend that the number of workers be made divisible by the number of CPUs in cases where they all perform approximately equal amount of work.

Finally, another contribution of our experiments is demonstrating that work-stealing can in certain scenarios perform worse than theoretical analyses indicate (see section 4.2.2), even with many workers and coarse-grained parallelism. The scatter/gather benchmark, an often-encountered type of workload, has proven to be especially elusive to efficient scheduling with work-stealing. A simple modification (WS-LAST) of the work stealing algorithm can bring significant performance benefits, without negative impact on other workloads.

### 5.3.3. Conclusions

In this section, we have evaluated load-balancing algorithms based on work-stealing and on graph partitioning. For the latter, we have adapted the algorithm of Devine

et al. [31, 32] to KPNs and augmented it with a run-time heuristic that triggers load-balancing when the idle time has exeeded a user-specified threshold. Algorithms based on graph-partitioning are usually applied to structured applications, such as large-scale simulations. Our work is the first evaluation of performance of such algorithms on less structured workloads, such as KPNs, that we are aware of.

In the first set of experiments (section 5.3.1), we have compared performance of the two algorithms on the synthetic workloads from section 5.2. This study has revealed that graph partitioning significantly increases locality of message traffic, which also decreases contention over channels. However, we found no correlation between locality and speedup. Furthermore, application running times under Devine's algorithm have very large variance, as illustrated in figure 5.25. This makes it clearly unsuitable as a general-purpose load-balancing algorithm on multi-core machines. Because of its inability to achieve consistent running times with the same workload, its use in distributed systems with unstructured workloads is also questionable, as it is hard to correlate resource allocation with expected speedup. We therefore recommend that the work-stealing be used as a general-purpose load-balancing algorithm on multi-core machines.

In the first set of experiments, we have also noticed that the achieved speedup under work-stealing was related to parallelism granularity. In the second set of experiments (section 5.3.2), we have further investigated this aspect. There, we have studied the original work-stealing algorithm (WS-CUR) and our modified work-stealing (WS-LAST), which we have described in section 4.2.2.4. In these experiments, we used the ring and pipeline benchmarks from section 5.2, and scatter/gather that we designed just for this experiment set. WS-CUR and WS-LAST have the same performance characteristics on all workloads, except ring and scatter/gather benchmarks. On ring benchmark, WS-CUR performs best, while on scatter/gather benchmark, WS-LAST performs best. On 8 CPUs, work-stealing achieves a speedup greater than 7 if each KP performs at least 100$\mu$s of work between it is unblocked and blocks again, and this result is independent of KPN topology. Alternatively, the same speedup is achieved when the number of steal attempts per seconds is held under $10^5$.

## 5.4. Summary

In this chapter, we have examined performance of applications when implemented in the KPN framework, low-level performance aspects of Nornir, as well as three scheduling strategies. For the latter two evaluations, we have designed and implemented a set of synthetic benchmarks which we have run in different Nornir configurations.

In all of our experiments, we have used a default channel capacity of 64 items. We did not investigate lower or higher values because of their predictable impact on performance. Lower values would increase scheduling and deadlock-detection overheads relative to the total running time of the program, which might nevertheless be justifiable in memory-constrained (embedded) systems.[13] Higher values would

[13]Though, in a memory-constrained system, channel capacities would usually be carefully designed

decrease overheads, so it would be harder to investigate their impact on the overall performance.

To evaluate application performance, we have used the word-count and k-means programs developed in chapter 3. We have compared their performance when running on top of Nornir and Phoenix [26], which is optimized for multi-core CPUs. Our experiments show that these applications run up to 2.7 and 1.7 times faster on Nornir than on Phoenix. The main reason for this performance improvement is that the KPN framework gives more flexibility in modeling application, which allows avoiding unnecessary work.

The evaluation of implementation options shows that the m:n scheduler combined with our optimized message-passing achieves better performance than the 1:1 scheduler in combination with either POSIX message queues or our optimzied transport. Our m:n scheduler is cooperative and performs context switch in user-space which significantly reduces scheduling overheads: the ring benchmark runs up to 6.5 times faster than with the 1:1 scheduler. Since communication is 1:1, it can be implemented more efficiently than the general m:n semantics of POSIX message queues: our zero-copy message transport has 1.7–1.9 less overhead than than POSIX message queues.

General performance evaluation of Nornir shows that the implementation scales well with the number of CPUs, and that benchmarks with enough available parallelism achieve almost an 8-fold speedup on 8 CPUs. Even though we have used mutexes to protect the runner's run-queues, we have noticed only very slight scalability problems due to contention. On the other hand, our results indicate that the centralized deadlock detection algorithm can severely degrade performance: our random graph benchmarks indicate that the maximum feasible rate is $\sim$ 50000 started deadlock detections per second on 8 CPUs. Thus, implementing a distributed deadlock detection mechanism, such as the one described in [22], is the first step towards improving Nornir. In the mean-time, negative impact of the centralized algorithm can be alleviated in two ways, both of which reduce the frequency of deadlock detections: by increasing initial channel capacities (they were set to 64 in our benchmarks), or by performing more work per message, which effectively decreases the total message throughput.

We have also evaluated performance of three dynamic load-balancing strategies: the original work-stealing algorithm (WS-CUR), work-stealing with our modification described in section 4.2.2 (WS-LAST) and the method based on graph partitioning (GP) described in section 4.2.3. These evaluations have been run with deadlock detection turned off in order to obtain more general results. The most important results of our evaluation can be summarized as follows:

- On a multi-core machine, work-stealing gives better speedup that GP for general-purpose workloads. The main problem with GP is its unpredictable performace, as demonstrated by figure 5.25.
- Work-stealing is sensitive to work division $d$ (amount of CPU time used per message), but also robust: performance degrades smoothly as $d$ moves from the

instead of set to a default value.

optimal value in either direction.

- WS-LAST and WS-CUR have practically the same performance characteristics, *except* on two specific topologies: ring (where WS-CUR performs better, and GP performs even better than WS-CUR) and scatter/gather (where WS-LAST performs better).
- The number of workers $n$ in the scatter/gather benchmark has a big influence on speedup: the best speedup is obtained when $n$ is divisible by the number of CPUs.

Our evaluation of the work-stealing algorithm thus complements the findings of Saha et al. [83] who have not tried to relate work division with potential scalability problems of the work-stealing algorithm.

# 6. Conclusion and further directions

To reduce power consumption, modern CPUs run at lower frequencies than their predecessors and compensate the reduced sequential performance with having more execution units on the same chip. Existing applications will thus run *slower* on new CPUs, unless they are redesigned to exploit the on-chip parallelism. Also, computing requirements of *new* applications are constantly growing – examples are games, multimedia transcoding and processing of large data amounts (stream processing). Parallel programming techniques, however, have developed at a much slower pace, and most parallel applications targeted towards workstation-class machines still use low-level and non-deterministic mechanisms such as threads, mutexes and condition variables.

Even though industry-backed solutions such as MapReduce [8] and Dryad [10] have been developed, they have two drawbacks in our context: they target large-scale distributed systems instad of multi-core machines, and they lack support for applications with feedback loops in their data-path. Being able to model feedback loops is a "must have" for a parallel programming framework as they appear quite often: the k-means application and H.264 benchmark are realistic examples used in this thesis. We have therefore proposed the use of Kahn process networks [12] (KPNs) as a one way of addressing the disparity between recent hardware and software developments. Even though KPNs offer a more flexible programming framework than existing frameworks, they do not incur increased complexity on the developer. Indeed, many of their desirable properties are also shared by the other frameworks:

- Determinism guarantees that the program will behave identically on each execution. Potential problems can therefore be reliably reproduced, diagnosed and fixed, which is an otherwise notoriously difficult with parallel and distributed applications.

- Processes share no state; the only means of communication through explicit message send and receive operations. This style of programming is analogous to developing applications handling ordinary files or network communication.

- Processes are written in the usual sequential style. Developers can therefore quickly become productive in the parallel programming domain using their existing knowledge.

- The assumptions underlying the KPN model are very general. Specifically, given a distributed KPN run-time, it would be possible to execute *unmodified* application in a parallel and distributed manner.

Since none of the existing KPN run-time implementations [20, 19, 15, 21, 22] satisfied our requirements stated in introduction, we have developed Nornir, a run-time environment for executing KPNs on multi-core machines. During implementation phase, we have focused on configurability, portability, simplicity, and performance.[1] These properties, together with collection of detailed run-time statistics, distinguish Nornir from the mentioned existing KPN frameworks. Generality, including support for feedback loops, and orientation towards efficiency on a single machine, distinguish Nornir from frameworks described in the related work, such as Google's MapReduce [8], Microsoft's Dryad [10], Cosmos with its programming language Scope [11] systems. IBM's System S with its programming language SPADE [13], which have been developed in parallel with our work, seem to be the only existing system whose features come close to Nornir. Very few details have been published about this system, and it is not available to the general public.

## 6.1. Review of problem statement and contributions

Our initial idea has been to investigate how KPNs can be used to address the limitations of existing frameworks. In section 1.2, we have therefore raised 4 research questions:

- How can we make an efficient run-time system for executing KPNs on multi-core machines?

  To answer this question, we have implemented Nornir, which is an implementation of a configurable, cross-platform, and efficient run-time environment for executing KPNs. (See chapter 4.) Nornir is modular so that message-passing, scheduling and load-balancing mechanisms can be replaced transparently to applications. Currently, Nornir offers the choice between 1:1 and m:n schedulers and three different load-balancing algorithms; its m:n scheduler scales well with the number of KPs and CPUs. We have measured that the cost of a single message send/receive operation combined with a context switch in a KPN with 1000 KPs takes $\sim 0.8\mu s$, which is comparable to other user-space scheduler implementations: Saha et al. [83] have measured the context switch cost of $0.748\mu s$ on a machine slightly faster than ours.

- Are KPNs applicable to parallelizing general-purpose applications and what is the required effort of doing so?

  To answer this question, we have thoroughly analyzed the MapReduce model and sample applications developed for Phoenix [26]. Our analyses have revealed that many applications do not need all steps performed by MapReduce, and that their performance could be improved by reimplementing them within the more flexible KPN framework. This is especially important in the context of running on one machine, where doing unnecessary work has much larger impact on performance

[1] Simplicity came first, though. We were guided by D. E. Knuth's maxim "[. . . ] premature optimization is the root of all evil."

than in a distributed scenario that MapReduce was designed for. We have also devised three ways of implementing the MapReduce semantics within the KPN framework.

- What are the performance characteristics of applications implemented within the KPN framework?

  To answer it, we have implemented the word-frequency and k-means applications, which are canonical MapReduce examples, within the Nornir KPN framework in two ways. The "natural" solution builds a KPN taking a starting point in the sequential program, while the "MapReduce" KPN solves the problem by exactly implementing the MapReduce semantics within the KPN framework. Our implementations and benchmarks clearly demonstrate the advantages of using the flexible KPN framework. Unlike MapReduce, the KPN model does not enforce the key-value paradigm, so the implementations of individual KPs for each problem resemble very much their sequential counterparts; the only differences are added communication statements. Our benchmarks show that the "natural" KPN solutions outperform their MapReduce solutions by a factor of 2.95 – 6.74 for the word-frequency application and by a factor of 1.17 – 1.54 on the k-means application. Our "natural" solutions also outperform their corresponding Phoenix implementations by factors of 2.65 – 2.76 and 1.28 – 1.75, respectively. The variation in speedup is due to different problem sizes used for benchmarking the applications.

- As part of answering the previous question, we have also designed a number of synthetic workloads with adjustable parallelism granularity. Given the right granularity, which depends on the workload in question, all benchmarks except H.264 have achieved an 8-fold speedup on 8 CPUs. The reason for the limited speedup of H.264 (which was 2.73) are data-dependencies in the KPN which limit the available parallelism in the KPN. These benchmarks, have also revealed a significant bottleneck in the current Nornir implementation – centralized deadlock detection algorithm. As a short-term solution, we have proposed to use larger default channel capacities (all our benchmarks used capacity of 64 messages); a long-term solution is implementing a distributed deadlock detection algorithm [22].

- What is the effect of KPN scheduling policies on application performance?

  To answer this question, we have implemented the work-stealing scheduling algorithm [30], and a load-balancing algorithm by Devine et al. [31, 32]. The latter algorithm is based on graph-partitioning and tries to reduce message traffic between KPs on different CPUs as well as KP migration. This algorithm is usually used for load-balancing structured computations in distributed systems, and we believe to be the first to evaluate it using unstructured workloads and multi-core machines. Our evaluations using synthetic benchmarks show that work-stealing has clear advantage over Devine's algorithm for scheduling workloads on multi-core machines: applications never achieved as good speedup as with

work-stealing, and in some cases it was much worse. We have also found a serious problem with their algorithm applied to our workloads: large variance of running times between consecutive runs of a single experiment, an aspect not discussed by the authors. This finding is of significance also in the context of distributed systems: it indicates that load-balancing based on graph partitioning, which is a key ingredient in Devine's algorithm, should be judiciously used with unstructured workloads.

- As a part of answering the previous question, we have also found a simple improvement to the original work-stealing algorithm. This improvement significantly benefits workloads of scatter/gather type, which are an important class of workloads, with no ill effects on other workloads. Our experimental results have shown that scatter/gather workloads can achieve worse speedup than predicted by the theoretical analysis of the work-stealing algorithm; this happens when the number of worker KPs is not divisible by the number of CPUs. We have also found that in this case, an *uneven* distribution of load might actually have a beneficial effect.

## 6.2. A critical review

A perfect work (masterpiece) is rarely found, and it is next-to-impossible to achieve perfection in the limited time available for fulfilling a PhD degree. We have identified certain limitations of our work, which addressing is left for future work.

We have run our benchmarks only machines with up to 8 cores. This, however, is not our choice, but a consequence of circumstances – we do not have machines with more cores in our lab.

We have chosen to using mutexes for protecting the run-queues in the work-stealing algorithm instead of using the non-blocking dequeue of Blumofe et al. [47]. Our choice was guided by our desire for simplicity, existing studies [83], and experience of others who claim that "[...] wait- and lock-free data structures are to be avoided as their failure modes are brutal (livelock is much nastier to debug than deadlock), their effect on complexity and the maintenance burden is significant, and their benefit in terms of performance is usually nil" [97]. Furthermore, using the lock-free dequeue would prevent us in implementing our modification to the work-stealing algorithm, so the performance comparisons would not be performed on the same grounds.

We do not recommend general-purpose use of our heuristics for determining when to rebalance the load for Devine's load-balancing algorithm. Depending on the workload, certain values of the idle time interval parameter $\tau$ can cause very frequent repartitionings and extreme degradation in performance. However, we had no existing work to guide us, and it is impossible to know whether an idea is good until it is actually implemented and tested. We are not aware of other attempts to apply Devine's and other load-balancing algorithms based on graph-partitioning to unstructured workloads on multi-core machines.

Many of our conclusions are based on synthetic workloads. The advantage of synthetic workloads is that they make it easy to accurately control the actual workload and to pinpoint potential performance problems. The disadvantage is that it is uncertain how well they model actual applications, and that they model only one aspect, namely CPU usage, while neglecting aspects such as memory access patterns. However, we have used also real workloads, and one of our synthetic workloads, H.264 encoder, bases its CPU usage patterns on data obtained by profiling.

## 6.3. Future directions

With our experimental studies, we have found and *quantified* some inefficiencies in Nornir, and proposed improvements in section 5.2.4. Implementing these improvements and evaluating their impact on performance, also on larger machines, is one future direction that addresses the first two of the above limitations.

We judge the overall results presented in this thesis as positive, and we are considering to extend Nornir to support distributed execution and use it in the context of the VERDIONE project [98], which requires a lot of computational resources for 3D video processing. Distributed execution opens at least two new challenges: extending channels to simultaneously support multiple transport mechanisms (e.g., sockets alongside of the existing mechanism), and implementing a distributed termination protocol [99]. For this application, we also envision a two-level load-balancing algorithm, where the upper level would place KPs across machines according to their communication and memory requirements, while the lower level would use work-stealing to schedule KPs within one machine.

We have also noticed that Devine's load-balancing algorithm does not faithfully model NUMA architectures. The algorithm assumes that a process migrates to a new node together with its data, but this is often not the case with applications executing on shared-memory NUMA machines. Developing models that take into account also the costs of a process accessing its own data is another direction for future research. Although not so important for our tests, such models will become more important as the number of cores in a computer system increases.

Our experimental evaluation of Devine's algorithm reveals a great variability of running times between consecutive runs of the same experiment, a result that is also significant for distributed systems. In this context, further investigation of Devine's algorithm with unstructured workloads is warranted because they have not discussed variance of running time in their publications. Also, its unpredictable performance makes it hard to estimate the application's completion time and to provision resources accordingly. Thus, we need dynamic load-balancing algorithms that balance load across CPUs and minimze inter-CPU (or, in distributed scenario, inter-machine) traffic, but yielding more predictable running times. Future research in this area includes two smaller problems: developing graph-partitioning algorithms that produce stable results, and developing better algorithms for detecting load-imbalance in an application on a small time-scale. These research directions would address the third of the above limitations.

Lastly, our investigation work-stealing algorithm performance has opened further questions. Our modification of work-stealing invalidates all stated reasons for using the LIFO policy for accessing the run-queues. It would thus be interesting to investigate whether changing this to FIFO would have any performance impact. Also, the experiments with the scatter/gather benchmark systematically achieve speedups lower than those predicted by existing theoretical analyses (see section 4.2.2.5). Thus, it would be interesting to repeat our experiments on machines with more cores to find the causes of these discrepancies and possibly extend existing analytical developments. These experiments would address the fourth of the above limitations.

# Bibliography


[1] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," Accessed July 2009, http://www-03.ibm.com/systems/p/hardware/whitepapers/power4.html.

[2] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–26, 2002.

[3] Sun Corporation, "Sun Niagara 2 Processor," Available online., Accessed July 2009, http://www.sun.com/processors/niagara/.

[4] J. Henry G. Baker and C. Hewitt, "The incremental garbage collection of processes," Cambridge, MA, USA, Tech. Rep., 1977.

[5] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, Windows XP update*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[6] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[7] P. B. Hansen, "Java's insecure parallelism," *SIGPLAN Not.*, vol. 34, no. 4, pp. 38–45, 1999.

[8] 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.

[9] 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.

[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*. New York, NY, USA: ACM, 2007, pp. 59–72.

[11] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, 2008.

[12] G. Kahn, "The semantics of a simple language for parallel programming." *Information Processing*, vol. 74, 1974.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: the system s declarative stream processing engine," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1123–1134.

[14] 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.

[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] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2004, pp. 220–229.

[17] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using kahn process networks: The compaan/laura approach," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 10340.

[18] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*. New York, NY, USA: ACM, 2000, pp. 13–17.

[19] 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

[20] 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.

[21] A. Olson and B. Evans, "Deadlock detection for distributed process networks," in *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, vol. 5, March 2005, pp. v/73–v/76 Vol. 5.

[22] 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.

[23] 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.

[24] 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.

[25] Željko Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, "Limits of work-stealing scheduling," in *To appear in: Proceedings of Job Scheduling Strategies for Parallel Processing*, 2009.

[26] 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.

[27] "Apache Hadoop," Accessed July 2009, http://hadoop.apache.org/.

[28] M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell BE Architecture," *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, vol. 1625, 2007.

[29] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.

[30] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[31] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek, "Parallel hypergraph partitioning for scientific computing," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, April 2006.

[32] 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.

[33] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, September 1972.

[34] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[35] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.

[36] NVIDIA Corporation, "CUDA Zone – The Resource for CUDA developers," Accessed July 2009, http://www.nvidia.com/object/cuda_home.html.

[37] Khronos Group, "OpenCL," Accessed January 2009, http://www.khronos.org/opencl/.

[38] "The openmp api specification for parallel programming," Accessed July 2009, http://openmp.org/wp/.

[39] D. J. B. Bil Lewis, *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall PTR, 1995.

[40] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[41] "Message passing interface forum," Accessed July 2009, http://www.mpi-forum.org/.

[42] 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*. New York, NY, USA: ACM, 2005, pp. 48–60.

[43] "GLib library," Accessed July 2009, http://www.gtk.org/.

[44] "Boost C++ libraries," Accessed July 2009, http://www.boost.org.

[45] Intel Corporation, "Threading building blocks," http://www.threadingbuildingblocks.org.

[46] P. A. Buhr and R. A. Stroobosscher, "The $\mu$ system: Providing light-weight concurrency on shared-memory multiprocessor computers running unix," *Software – Practice and Experience*, vol. 20, no. 9, pp. 929–964, 1990.

[47] 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.

[48] Intel Corporation, "Threading building blocks reference manual," document no. 315415-001US, rev.1.13.

[49] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1997.

[50] C. Fournet and G. Gonthier, "The reflexive cham and the join-calculus," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 372–385.

[51] N. Benton, L. Cardelli, and C. Fournet, "Modern concurrency abstractions for c#," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 5, pp. 769–804, 2004.

[52] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: why is it only a research toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[53] P. E. McKenney, M. M. Michael, and J. Walpole, "Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory," in *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*. New York, NY, USA: ACM, 2007, pp. 1–5.

[54] AMD Corporation, "Advanced synchronization facility proposal," http://developer.amd.com/CPU/ASF/Pages/default.aspx.

[55] J. Armstrong, "A history of erlang," in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA: ACM, 2007, pp. 6–1–6–26.

[56] SGS-THOMSON Microelectronics Limited, "occam 2.1 reference manual," Accessed July 2009, http://www.wotug.org/occam/documentation/oc21refman.pdf.

[57] P. Welch and F. Barnes, "Communicating mobile processes: introducing occam-pi," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, A. Abdallah, C. Jones, and J. Sanders, Eds., vol. 3525. Springer Verlag, April 2005, pp. 175–210, to appear. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2005/2162

[58] S. A. Edwards and O. Tardieu, "SHIM: a deterministic model for heterogeneous embedded systems," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2005, pp. 264–272.

[59] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[60] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[61] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, i," *Inf. Comput.*, vol. 100, no. 1, pp. 1–40, 1992.

[62] T. Lee, E.A.; Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[63] J. Larson, "Erlang for concurrent programming," *Queue*, vol. 6, no. 5, pp. 18–23, 2008.

[64] R. Lämmel, "Google's mapreduce programming model — revisited," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 208–237, 2007.

[65] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, 2005.

[66] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler, "A history of haskell: being lazy with class," in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA: ACM, 2007, pp. 12–1–12–55.

[67] R. McDougall and J. Mauro, *Solaris Internals(TM): Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition)*. Prentice Hall PTR, 2006.

[68] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4BSD operating system*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996.

[69] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, Berkeley, CA, USA, 1995. [Online]. Available: http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/

[70] N. Lynch and E. W. Stark, "A proof of the kahn principle for input/output automata," *Information and Computation*, vol. 82, pp. 81–92, 1989.

[71] A. A. Faustini, "An operational semantics for pure dataflow," in *Automata, Languages and Programming*. Springer-Verlag, 1982, pp. 212–224.

[72] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," *Information Processing*, vol. 77, 1977.

[73] 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.

[74] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, 2007.

[75] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 29–43.

[76] 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)*. New York, NY, USA: ACM, 2007, pp. 1029–1040.

[77] "Rhino: Javascript for java," Accessed July 2009, http://www.mozilla.org/rhino/.

[78] Silicon Graphics, Inc., "Standard template library programmer's guide," Accessed July 2009, http://www.sgi.com/tech/stl/.

[79] R. S. Engelschall, "Portable multithreading: the signal stack trick for user-space thread creation," in *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 20–20.

[80] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2003, pp. 4–4.

[81] Valgrind developers, "Valgrind," Available online., http://valgrind.org/info/tools.html.

[82] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," Cambridge, MA, USA, Tech. Rep., 1996.

[83] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling scalability and performance in a large scale cmp environment," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 73–86, 2007.

[84] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, Jun. 1998, pp. 212–223, proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[85] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1996, pp. 267–275.

[86] D. Lea, "A java fork/join framework," in *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. New York, NY, USA: ACM, 2000, pp. 36–43.

[87] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. New York, NY, USA: ACM, 2007, pp. 15–23.

[88] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 7, pp. 673–693, 1999. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=780863

[89] L. McVoy and C. Staelin, “lmbench: portable tools for performance analysis,” in *ATEC ’96: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23.

[90] I. E. G. Richardson, “H.264/mpeg-4 part 10 white paper,” Available online., http://www.vcodex.com/files/h264_overview_orig.pdf.

[91] VideoLAN team, Available online., http://www.videolan.org/developers/x264.html.

[92] H. Espeland, *Investigation of parallel programming on heterogeneous multiprocessors. Master thesis.* Oslo: The University of Oslo, 2008. [Online]. Available: http://urn.nb.no/URN:NBN:no-19898

[93] L. Hung and S. Sakai, “Dynamic estimation of task level parallelism with operating system,” in *Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, Dec. 2005, pp. 6 pp.–.

[94] 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.

[95] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *Annals of Mathematical Statistics*, 1947.

[96] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *Proceedings of the symposium on Operating systems design and implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2008.

[97] B. Cantrill and J. Bonwick, “Real-world concurrency,” *Queue*, vol. 6, no. 5, pp. 16–25, 2008.

[98] “Verdione project,” Accessed July 2009, http://verdione.org/.

[99] N. Francez, “Distributed termination,” *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 42–55, 1980.

[100] R. Diestel, *Graph Theory, 3rd ed.* Springer-Verlag, Heidelberg, 2005.

[101] O. Goldschmidt and D. S. Hochbaum, “A polynomial algorithm for the k-cut problem for fixed k,” *Math. Oper. Res.*, vol. 19, no. 1, pp. 24–37, 1994.

[102] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, “The complexity of multiway cuts (extended abstract),” in *STOC ’92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1992, pp. 241–251.

[103] P. Elias, A. Feinstein, and C. Shannon, "A note on the maximum flow through a network," *Information Theory, IEEE Transactions on*, vol. 2, no. 4, pp. 117–119, Dec 1956.

[104] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[105] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[106] Y. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Mathematics-Doklady*, vol. 11, no. 5, pp. 1277–1280, 1970.

[107] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, 1988.

[108] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," in *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1981, pp. 114–122.

[109] M. Stoer and F. Wagner, "A simple min-cut algorithm," *J. ACM*, vol. 44, no. 4, pp. 585–591, 1997.

[110] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[111] H. Saran and V. V. Vazirani, "Finding $k$ cuts within twice the optimal," *SIAM J. Comput.*, vol. 24, no. 1, pp. 101–108, 1995.

[112] R. E. Gomory and T. C. Hu, "Multi-terminal network flows," *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, no. 4, pp. 551–570, 1961. [Online]. Available: http://www.jstor.org/stable/2098881

[113] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 1, pp. 291–307, 1970.

[114] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *DAC '82: Proceedings of the 19th conference on Design automation*. Piscataway, NJ, USA: IEEE Press, 1982, pp. 175–181.

[115] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is NP-hard," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 153–159, 1992.

[116] G. Even, "Fast approximate graph partitioning algorithms," *SIAM J. Comput.*, vol. 28, no. 6, pp. 2187–2214, 1999.

[117] K. Andreev and H. Räcke, "Balanced graph partitioning," in *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2004, pp. 120–124.

[118] U. Feige and R. Krauthgamer, "A polylogarithmic approximation of the minimum bisection," *SIAM J. Comput.*, vol. 31, no. 4, pp. 1090–1118, 2002.

[119] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983. [Online]. Available: citeseer.ist.psu.edu/kirkpatrick83optimization.html

[120] T. N. Bui and B. R. Moon, "Genetic algorithm and graph partitioning," *IEEE Trans. Comput.*, vol. 45, no. 7, pp. 841–855, 1996.

[121] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–13.

[122] R. Baños, C. Gil, J. Ortega, and F. G. Montoya, "Multilevel heuristic algorithm for graph partitioning," in *3rd European Workshop on Evolutionary Computation in Combinatorial Optimization*, vol. 2611. Springer, 2003, pp. 143–153.

[123] R. Battiti and A. A. Bertossi, "Greedy, prohibition, and reactive heuristics for graph partitioning," *IEEE Trans. Comput.*, vol. 48, no. 4, pp. 361–385, 1999.

[124] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, 2008.

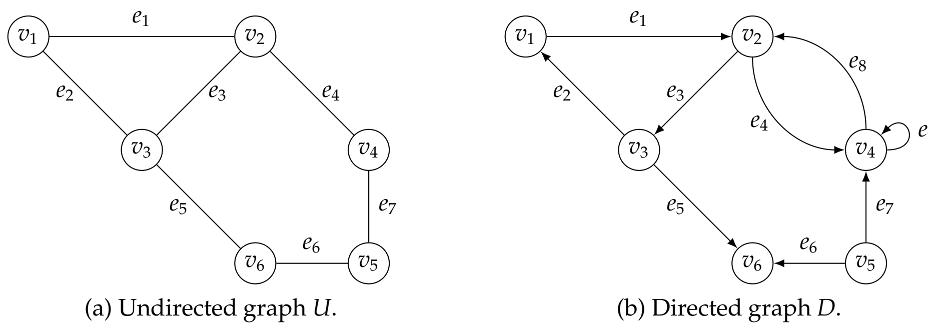
[125] AMD Corporation, "AMD64 architecture programmer's manual volume 2: System programming," document no. 24593, rev.3.14.

[126] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

# A. Review of graph theory concepts

To make the thesis self-contained, we give a short and rather informal overview of elementary terms from graph theory. A much more comprehensive and rigorous introduction may be found in [100].[1]

An *undirected graph* $G$ is a pair $G = (V, E)$ where $V$ is the set of *vertices* and $E$ is a set of *edges*, where each edge connect exactly two vertices. Formally, an edge $e \in E$ is a two-element subset of $V$, i.e., $e = \{v_j, v_k\}$ for some $v_j, v_k \in V$ with $v_j \neq v_k$. For example, the graph in figure A.1a has $V = \{v_1, v_2, \ldots, v_6\}$ and $E = \{e_1, \ldots, e_7\} = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_6\}, \{v_5, v_6\}, \{v_4, v_5\}\}$.




(a) Undirected graph $U$. (b) Directed graph $D$.

Figure A.1.: Examples of graphs.

A *directed graph* $G$ is defined similarly as undirected graph, except that $E \subseteq V \times V$, i.e., each edge is an *ordered* pair of vertices $(v_1, v_2)$ with $v_1, v_2 \in V$; $v_1$ is the *source* vertex and $v_2$ is the *target* vertex of the edge. For the graph in figure A.1b, $V = \{v_1, v_2, \ldots, v_6\}$, $E = \{e_1, \ldots, e_{10}\} = \{(v_1, v_2), (v_3, v_1), (v_2, v_3), (v_2, v_4), (v_3, v_6), (v_5, v_6), (v_5, v_4), (v_4, v_2), (v_4, v_4)\}$. The directedness of the graph is represented by the arrows from the source to the target vertex on the edges.

An undirected graph may be represented as a directed graph where each undirected edge is represented as two edges in opposing directions in the directed graph. For example, $e_4$ from graph in figure A.1a would be transformed into edges $e_4$ and $e_8$ in the directed graph.

A *path* in a graph from vertex $s$ to vertex $e$ is an alternating list of vertices and edges, starting with $s$ and ending with $e$. If the graph is directed, then edges must be followed in their directions. For example, $v_6, e_6, v_5, e_7, v_4$ is a path from $v_6$ to $v_4$ graph $U$, but *not*

[1] The book is also available online at `http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/index.html`

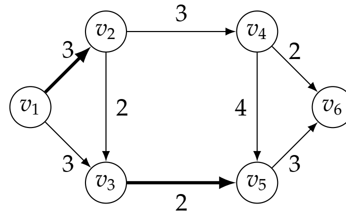



Figure A.2.: Example of a minimal-weight cut separating $v_1$ and $v_6$. The cut consists of edges $(v_1, v_2)$ and $(v_3, v_5)$ (thick lines) and has weight of $3 + 2 = 5$. The vertex set is partitioned into parts $S = \{v_1, v_3\}$ and $T = \{v_2, v_4, v_5, v_6\}$.

in graph $D$ because there is no directed edge from $v_6$ to $v_5$. A *loop* is an edge which connects the vertex to itself ($v_4$).

A *weighted graph* has associated weights with edges, vertices, or both, i.e., weight is a function $w_S : S \to \Re$, where $S$ is $V$ or $E$. Weight of the path is the sum of weights of edges within the path. Weight of a subset of vertices $V' \subseteq V$ is the sum of weights of individual vertices in $V'$.

Given a (directed or undirected) graph $G = (V, E)$ and $s, t \in V$, an *s-t cut* is a subset of edges $C \subset E$ whose removal disconnets the vertices $s$ and $t$. In other words, after removing edges in $C$, the graph will have been partitioned into two *components* with vertex sets $S$ and $T$, such that $s \in S$, $t \in T$ and such that there does not exist a path from any vertex in $S$ to any vertex in $T$, or vice-versa.

If the graph has weighted edges, a problem of particular importance for us is finding an $s$-$t$ cut $C$ of *minimal weight*, i.e., a cut that minimizes the sum $\sum_{e_i \in C} w_e(e_i)$; note that there may be several cuts with the same minimal weight. Figure A.2 shows an example of a minimal-weight cut in a directed graph.

The concepts described for undirected graphs can be generalized to *hypergraphs* and *hyperedges*. Formally, a *hypergraph* $HG = (V, H)$ consists of the set of vertices $V$ and set of hyperedges $H$. Whereas an edge in a graph is a vertex subset with size of *exactly 2*, each hyperedge $h \in H$ in a hypergraph is a vertex subset of arbitrary size, i.e, $h \subseteq V$ and $1 \leq h \leq |V|$; this is awkward to visualize and we shall not attempt to do so in here. All concepts described for graphs so far can be directly carried over to hypergraphs. Due to their greater generality, there are several metrics which can be used to evaluate the cost $c$ of a cut; in this work we use the $(k-1)$-metric:

$$c = \sum_{h_i \in H} w_i(\lambda_i - 1)$$

where the sum is evaluated over all hyperedges, $w_i$ is the weight assigned to hyperedge $h_i$ and $\lambda_i$ is the number of different parts that the hyperedge spans. Note that the hyperedge's contribution to $c$ is 0 if all of its vertices are contained in the same partition.

# B. Graph cuts and partitions

In section B.1 we describe approaches for solving the k-cut problem, which reduces communication across partitions, but does attempt to make the vertex partition balanced, i.e., such that all partitions are of approximately equal weight. In section B.2 we review graph bisection algorithms, which partition the graph into *two* parts of prespecified weight ratio and reduced cut size. In section B.3 we show how a bisection algorithm can be used to partition a graph into $k$ parts. Finally, in section B.4 we discuss a graph mapping problem which is used in contexts wit nonuniform communication costs between pairs of nodes in a distributed system.

## B.1. The k-cut problem

The $k$-cut problem may be formally defined as follows. Given an undirected graph $G = (V, E)$ with non-negative edge weights, and an integer $k$, find a subset of edges of minimum weight, $S \subseteq E$, whose removal leaves a graph with $k$ connected components. This problem is NP-hard for general graphs and arbitrary $k$, but polynomial algorithms are known for fixed $k$:

- For general graphs an infeasible, though polynomial, algorithm of complexity $O(n^{k^2} T(n, m))$ is described in [101], where $T(n, m)$ is the running time required to find minimum $(s, t)$ cut on a graph.
- An exact algorithm for *planar* graphs may be found in [102]; its complexity is $O(n^{ck})$ for some constant $c$.

In the above descriptions we have used $n = |V|$ and $m = |E|$; these short-hands shall be used throughout this section.

### B.1.1. Maximum flow and (2-)cut of a graph

Given two vertices $s$ and $t$, a *cut* of the graph $(V, E)$ is a subset of edges whose removal partitions the graph into two components $S \subset V$ and $T \subset V$ such that $s \in S$ and $t \in T$. The "max-flow, min-cut" theorem [103, 104] states that the maximum amount of flow through the network is equal to the capacity of a minimal cut. Consequently, an obvious way of finding a minimal cut is to find the maximum flow in the graph.

The first maximal-flow algorithm was developed by Ford and Fulkerson [104] and has a complexity of $O(mf)$ where $f$ is the value of the maximal flow; note that the algorithm is *not* polynomial because the running time depends on $f$. Polynomial-time algorithms have followed: $O(n^3)$ algorithm of Edmonds and Karp [105], Dinic's $O(mn^2)$

algorithm [106], and $O(n^3)$ algorithm of Tarjan and Goldberg [107]. The running time of the latter algorithm may be improved to $O(nm\log(n^2/m))$ by using a dynamic tree data structure [108].

Recent research has developed algorithms that directly find the cut of a graph, thus avoiding an intermediate max-flow computation. Stoer and Wagner [109] present a simple minimum-cut algorithm with complexity of $O(nm + n^2 \log n)$ when implemented with the help of a Fibonacci heap [110].

### B.1.2. Saran-Vazirani algorithm

Two simple and greedy approximation algorithms, “efficient” and “split”, for computing a $k$-cut of a graph are given in [111]. Both algorithms find a cut whose weight is within a factor of $2 - 2/k$ of the optimal. By constructing a Gomory-Hu tree [112] using the Tarjan’s max-flow algorithm, which is executed $n - 1$ times, the “efficient” algorithm achieves a running time of $O(mn^2 + n^{3+\epsilon})$ for any $\epsilon > 0$. Their “split” algorithm requires $O(kn)$ cut computations to find a $k$-cut.

## B.2. Graph bisection

The goal of a graph bipartitioning (or bisection) algorithm is to partition the set of vertices into two sets of given weight ratio, such that the cut weight is minimized.

### B.2.1. KL heuristics

The first, and still one of the most cited bisection algorithms is that of Kernighan and Lin (KL) algorithm [113], with complexity of $O(n^2 \log n)$. The algorithm is designed for graphs where all vertices have the same (unit) weight, and the quality of the approximated bisection is investigated empirically.

The algorithm is also applicable to graphs having vertices of weight $w > 1$. In this case, each such vertex is converted to a cluster of $w$ vertices of unit weight. These vertices are then connected by edges of sufficiently high cost to form a complete graph, which prevents them from being assigned to different parts. Obviously, this approach can significantly increase the running time of the algorithm, since the complexity becomes $O(W^2 \log W)$ where $W$ is the sum of all vertex weights.

To partition the graph into $k > 2$ parts, the algorithm is repeatedly applied to pairs of vertex subsets starting from *some* initial partitioning into $k$ parts. This increases the algorithm’s complexity since there are $\binom{k}{2}$ pairs of subsets. Furthermore, this method is limited to swapping vertices between two subsets at a time, whereas much better results could be obtained by swapping vertices between 3, 4, or more sets at a time.

### B.2.2. FM heuristics

Fiduccia and Mattheyses (FM) heuristics [114] is an *almost* linear-time variant of the KL-heuristics with running time $O(Mn)$, $M$ being the largest vertex weight in the graph.[1] Unlike the KL-algorithm, which is based on swaps, FM-heuristics is based on vertex moves. In addition, the algorithm is defined for hypergraphs, and it can handle vertices of weights $w > 1$ without the need for introducing extra vertices.

The data structure used by the algorithm as presented by the authors is an array of linked lists, with the array being indexed by vertex weight. Searching for the next-lightest vertex, a step often executed by the algorithm, is therefore an $O(M)$ operation. Thus, the algorithm is slowed down in in presence of large vertex weights, but by a smaller factor than the KL heuristics.

The array can be replaced by a balanced tree sorted by vertex weights. In this case, the search for the next-lightest vertex is logarithmic in the number of different vertex weights. Such data structure will also yield significant space savings when weights are a sparse subset of the set $\{1, 2, \ldots, M\}$.

## B.3. The k-partition problem

The graph partitioning problem may be formally defined as follows. Given an undirected graph $G = (V, E)$, where both edges and vertices have non-negative weights, and an integer $k$, partition the vertex set into $k$ disjoint subsets $S_i \subset V, 1 \leq i \leq k$ with the following objective:

- The weights of $S_i$ should be as close to each other as possible, i.e.,

$$\sum_{1 \leq i < j \leq k} |w(S_i) - w(S_j)|$$

  should be minimized (it is also said that the partition is *balanced*).

- The total cost of the cuts

$$\sum_{1 \leq i < j \leq k} C_{ij}$$

  shold be minimized, where $C_{ij}$ is the cost of the cut between $S_i$ and $S_j$.

This problem is also NP-hard for general graphs, and it is, in a sense, even harder than the $k$-cut problem because even finding good approximation solutions, also for planar graphs, is NP-hard [115]. Therefore, a number of heuristics[2] have been developed to tackle the problem.

[1] Technically, it is trivial to modify the algorithm so that the running time is proportional to the *difference* between the largest and smallest vertex weight.

[2] Approximation algorithms guarantee an upper bound on the discrepancy between the quality of approximate and optimal solution. Heuristic methods provide no such guarantees.

### B.3.1. Theoretical bounds

As already mentioned, even finding good approximations to the graph partitioning problem is NP-hard. The problem is first generalized to finding a $(k, \nu)$ partition where the graph is partitioned into $k$ pieces of size at most $\nu \frac{n}{k}$, while minimizing the edge weight between partitions.

The work of Even [116] defines a new problem, finding minimum capacity $\rho$-separators, that can be used for solving the partitioning problem. Their algorithm focuses on the case $\nu \geq 2$ and it achieves an approximation ratio of $O(\log n)$.

A recent work [117] considers the $(k, 1 + \epsilon)$ partitioning problem for arbitrary $\epsilon$, where two significant results are derived:

- The $(k, 1)$ partitioning problem has no polynomial-time approximation within finite approximation factor, unless P=NP.
- An algorithm which finds a partitioning where each component contains at most $(1 + \epsilon)\frac{n}{k}$ nodes and cut weight at most $O(\log^2 n/\epsilon^4)$ within the optimal.

However, for constant $k$, efficient approximation algorithms may exist, and one for the $(2, 1)$ partitioning problem is presented in [118]; their approximation is within $O(\log^2 n)$ within the optimal.

Saran and Vazirani [111] have also developed an algorithm for finding an approximate solution to the graph bisection problem, whose running time is dominated by the time needed to find the Gomory-Hu tree in the graph, which needs $n - 1$ cut computations. Their approximation is within the factor of $n/2$ of the optimal solution, and when extended in “a straightforward manner” to any fixed $k$, the approximation is within the factor of $\frac{k-1}{k}n$ within the optimal solution. Thus, as $k$ grows, the quality of the approximation worsens with the size of the graph.

### B.3.2. Recursive bisection

By recursively applying a graph bisection algorithm with suitably specified ratios at each step, a graph can be partitioned into any (not necessarily power of two) number of parts. Figure B.1 exemplifies partitioning the graph into 5 parts via recursive bisection. Each node in the tree represents a single partition and it is labeled with the fraction of the total graph weight that it contains. Internal tree nodes represent applications of the bisection algorithm, while the external nodes represent the final partitions.

### B.3.3. Stochastic methods

*Simulated annealing* is an optimization method based on concepts of statistical mechanics which postulate that a system tends to stabilize in a state having minimal energy. The algorithm is in detail described in [119], where the authros show how it can be applied to hard combinatorial optimization problems such as chip placement, wire routing and the traveling salesman problem.

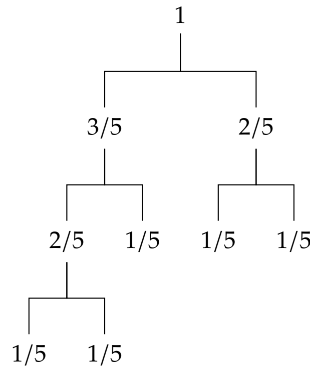



Figure B.1.: Using recursive bisection to partition the whole graph, represented by the root node, into 5 parts.

*Genetic algorithms* are based on the evolutional concept of "survival of the fittest". The problem is encoded into "gene sequences" which are then crossed and mutated over many iterations. Bui and Moon [120] apply a hybrid genetic algorithm to the graph partitioning problem and evaluate their approach on many types of graphs. Hybrid genetic algorithms apply local improvement algorithms after each iteration; the authors use a variant of Kernighan-Lin heuristics for that purpose.

The common feature of both algorithms is that they are probabilistic, so it is hard to analyze their running time. Some timing measurements are given in [120] and they indicate that the genetic algorithm would be too slow to be used for dynamic reconfiguration of a running process network on a single machine.

### B.3.4. Multilevel and greedy heuristics

Multilevel approaches [121, 122] first group vertices to build clusters which become the vertices of a coarse graph. This process is repeated until a small enough graph is obtained on which an existing partitioning procedure is applied. The graph is then expanded until its original configuration, and at each step a refinement is applied. Multilevel approaches are applicable to the generalized partitioning problem where vertices are assigned a *vector* of weights, and partition weights are balanced with respect to each weight.

Battiti and Bertossi [123] propose new heuristics for graph bisection, evaluate their performance in detail, and compare with (at the time) state of the art libraries, such as MeTis, Chaco, Scotch or Party. The min-max greedy heuristic proposed in the paper executes on graphs with up to 5000 vertices in under 0.1 second of CPU time in most cases, but it can take up to 0.8 seconds in rare cases. The other libraries execute the same test in under 0.05 seconds, but the running time can increase up to 0.6 seconds in rare cases. The tests have been run on AlphaServer 2100 with four 250 MHz CPUs (the programs tested are not parallel, i.e., they were using only a single CPU).

## B.4. Mapping problem

We have so far introduced the $k$-cut and $k$-partition problems. Heuristics that solve the $k$-partition problem can be used for architectures where the communication cost between any pair of CPUs has the same cost. However, there exist architectures, so-called NUMA (non-uniform memory architecture), where this assumption does not hold. In a NUMA system, each CPU (also called node) has some local memory, access to which takes least time. Remote memory access, i.e. access to local memory of another node, takes time proportional to the number of links connecting the two nodes. An example of such architecture is AMD64 with HyperTransport bus, where $2^n$ CPUs are connected into a $n$-dimensional hypercube (see figure C.1 for $n = 2$).

After the tasks of the original process graph have been assigned to individual CPUs, a link $p$ between any pair of processes in the original process graph will have been mapped to a link $p'$ in the target architecture graph. The goal is to place intensively communicating processes onto nearby nodes, while preserving load-balance criterion at the same time. If the target architecture graph is a complete graph on $k$ nodes where all nodes and all edges have equal individual weights, the mapping problem simplifies to the $k$-partition problem discussed in the previous section. Such problem is solvable, for example, by dual recursive bipartitioning as implemented in the Scotch library [124].

# C. Computer architecture

There are three architecture factors that are visible to software in functional or non-functional way, and which considerably increase programming complexity of modern computer systems: the CPU's memory consistency model, its cache hierarchy, and possible NUMA architecture of the whole computer. These factors also blur the line between shared-state and message-passing concurrency.

## C.1. Memory subsystem

In a NUMA system (see figure C.1), each memory access is either *local* or *remote*. Local memory bank(s) are directly accessible through the CPU's data and address busses, so access to them is fastest. Accesses to memory banks attached to other CPUs (remote accesses) are transparently converted by the hardware into messages that are forwarded across the interconnect bus. Thus, latency and bandwidth of remote memory accesses is dependent on the *path length* between the initiating CPU and the target memory bank, as well as current link utilizations along the path.

In addition, each CPU has a small amout of *cache memory*, organized in hierarchical levels, which is many times faster than dynamic RAM. The level one (L1) cache is smallest and fastest, often matching the speed of the CPU itself. If multiple CPUs have a copy of the sama data piece in their caches, and some CPU modifies that data, the other caches have to be *invalidated*. Early NUMA systems did not perform automatic

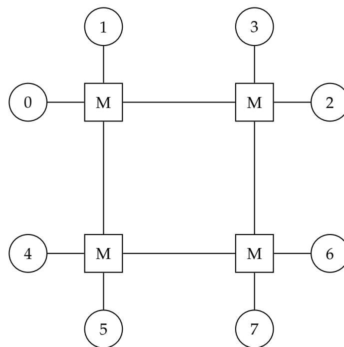



Figure C.1.: Example of a NUMA topology. M denote memory nodes, and 0–7 are are CPUs in the system. Assuming that all communication paths have equal bandwidth and latency, it takes CPU 0 twice as long to access 2's local memory and three times as long to access 6's local memory than its own local memory.

invalidation, which made them simpler to build, but more complex to program since *cache coherency* protocols had to be implemented in software. Cache-coherent NUMA architectures (ccNUMA) employ special protocols in hardware, e.g. MOESI [125], to ensure that all CPUs see consistent memory state at all times. This greatly simplifies the programming model, but also complicates the design. The work in this thesis is aimed towards ccNUMA architectures, more specifically, towards the widely available systems with AMD Opteron CPUs.

Modern CPUs introduce additional difficulty by introducing *relaxed memory consistency model* [126]. To optimize memory accesses, they can *buffer* memory writes and/or *reorder* memory reads and writes, which can cause incorrect functioning of various algorithms on machines with multiple CPUs. The needed memory ordering is achievable by using *memory barrier* instructions, which ensure that memory reads and writes are performed in the program order.

The strongest consistency model is *sequential consistency* where all CPUs in the system see changes to shared memory locations in the program order. The 64-bit AMD architecture which we used for our experiments uses *processor consistency*, which is a slightly weaker model.

Many operating systems implement *virtual memory* which gives several benefits:

- Protection and isolation between processes,
- Illusion of larger memory than is physically installed,
- Reduced memory usage by sharing code and data, where applicable.

Mapping from *virtual addresses* to *physical addresses* is defined by page tables, which are set up by the operating system; the mapping is at run-time performed by the memory management unit (MMU). Since main memory is slow and address translation must be performed for *every* memory access, MMU caches recent address translations in a small dedicated cache called translation lookaside buffer (TLB). If a process accesses a location that is not cached in a TLB, a TLB miss occurs. TLB misses are usually handled automatically by hardware, and their cost is even more expensive than a data cache miss. The performance degradation caused by TLB misses also limits the performance of the OS scheduler, since the TLB must be flushed on each context switch to another process. On CPU architectures that have *tagged address spaces*, the performance impact of TLB flush is somewhat reduced.

## C.2. Performance effects of memory organization

To experimentally evaluate effects of memory organization on performance, we have performed a series of tests with different workloads. The experiments have been performed on an otherwise idle 2.6 GHz 4-CPU AMD Opteron machine, with 2 cores on each CPU, and 64 GB of RAM running linux kernel 2.6.27.3. The benchmark program is single-threaded and has been compiled as 64-bit with GCC 4.3.2 and maximum optimizations turned on (`-m64 -O3 -march=opteron`).

The program is configurable to perform operations over a block of memory in either sequential or random order. More specifically, the following parameters are configurable:

- Memory block size (fixed to 1 GB in all experiments).
- NUMA policy (allocation from a specified node, or interleaved over all nodes).
- Operation to perform (byte increment or AES encryption in ECB mode).
- Order of operations (sequential or randomized).

The program binds itself to run on node 0,[1] allocates the specified amount of memory, and fills it with zeros before starting the time measurement, which is necessary to eliminate the effects of demand-paging. In the initial experiments, without prefilling the memory block with zeros, we have measured that demand paging has prolonged execution by an additional 1 second spent in the kernel. Filling the memory block with zeros before starting the benchmarking loop has reduced the loop's system time usage to zero.

Then, since AES block size is 16 bytes, an identity permutation of $2^{30}/16 = 2^{26}$ elements is generated; the whole array takes 256MB and is accessed sequentially. If randomization has been selected, the array is permuted by the C++ standard library function `random_shuffle`. Each program run always uses the same random seed, which in turn generates the same random permutation.

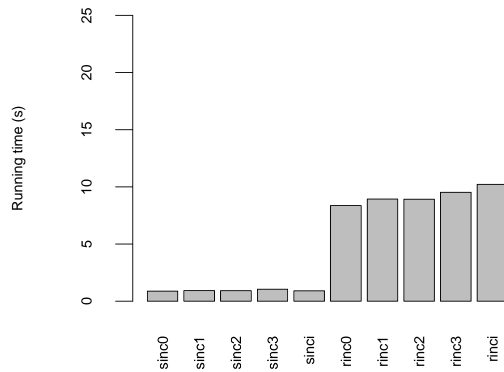
The benchmarking loop operates over blocks of 16 bytes. In case of AES, the block is ECB-encrypted in place. In case the increment operation has been selected, each byte within the block is sequentially accessed and incremented by one in place.

Benchmark results are shown in figure C.2. The biggest difference for both operations is between sequential and random access. Sequential access is automatically detected by the hardware and triggers automatic data prefetching which hides any additional latencies caused by the NUMA architecture.

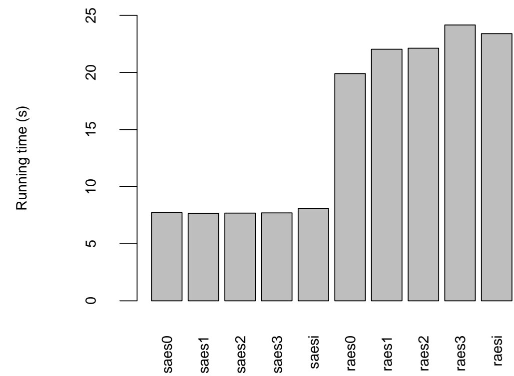
With random accesses, every access to a new 16-byte block must fetch data from RAM, which greatly increases execution time because RAM is many times slower than the CPU. Compared to the best case (execution on node 0 and sequential access pattern), the increment test became slower by a factor of 9.5 (memory on node 0) to 11.24 (interleaved allocation), and the AES test became slower by a factor of 2.6 (memory on node 0) to $\sim 3.1$ (memory on node 3). Thus, latency of access to remote memory increases by $\sim 9\%$ for each additional "hop".

We initially expected that interleaved allocation will have no effect on performance for sequential accesses, and somewhat better performance than that of the worst case for random accesses (program executing on node 0 and accessing memory on node 3) because only 1/4 of all memory accesses will be to node 3. However, we can see that, contrary to the expectation, interleaved allocation has *increased* running time both for the increment test with random access pattern as well as for the AES test with sequential

[1]That is, it may execute on any of the two cores.




(a) Byte increment




(b) AES encryption

Figure C.2.: Execution time for sequential and randomized accesses, with memory fixed on some node, or interleaved over all nodes. Access pattern is either (s)equential or (r)andom, operation is (inc)rement or (aes), and the memory is either allocated from a single node (0-3) or (i)nterleaved over all nodes.

access pattern. Since detailed study of computer architecture is out of the scope of this thesis, we did not attempt to find the cause of these phenomena.

# D. A complete code example

For completeness, we give a complete listing of a sample Nornir application. The following program implements the KPN-FLEX variant of the k-means benchmark described in section 3.3.3.

```
#include <stdlib.h>
#include <string.h>
#include <algorithm>
#include <iostream>
#include <string>
#include <boost/bind.hpp>
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include "kahn.h"

using std::cin;
using std::cerr;
using std::clog;
using std::endl;
using boost::get;

static const int  GRIDSIZE = 1000;
static const int  DIM = 3;

// Last element is count/class
struct point
{
  int v[DIM];
  int c;
};

static const point ZEROPOINT = { {0, 0, 0}, 0 };

std::ostream &operator<<(std::ostream &os, const point &p)
{
  os << '[';
  for(size_t i = 0; i < DIM; ++i)
    os << p.v[i] << ',';
  os << p.c << ']';
  return os;
}
```

```
typedef std::vector<point> point_vec;
typedef boost::tuple<point_vec::iterator,
  point_vec::iterator> point_range;

///////////////////////////////////////////////////////////////////////////
// PRODUCE/ITERATE.

class producer : public perun::actor
{
  point_vec points_;
  point_vec means_;

  void  fill(point_vec &pv, size_t n);
  void  distribute_means();
  void  partition();
  size_t  iterate();

protected:
  virtual void behavior();

public:
  std::vector<perun::send_port<point_range>*> out1;
  std::vector<perun::send_port<point>*>   out2;
  std::vector<perun::recv_port<point>*>   in;

  producer(size_t nworkers, size_t npoints, size_t nmeans);
};

producer::producer(size_t nworkers, size_t npoints, size_t nmeans)
{
  for(size_t i = 0; i < nworkers; ++i) {
    out1.push_back(new perun::send_port<point_range>(this));
    out2.push_back(new perun::send_port<point>(this));
    in.push_back(new perun::recv_port<point>(this));
  }
  fill(points_, npoints);
  fill(means_, nmeans);
}

void producer::behavior()
{
  partition();
  size_t niter = iterate();

  for(size_t i = 0; i < out2.size(); ++i)
    out2[i]->set_eof();

  for(size_t i = 0; i < means_.size(); ++i)
```

```
    clog << means_[i] << endl;
  clog << "NITER=" << niter << endl;
}

// Partition the work and send out initial means.
void producer::partition()
{
  size_t chunk = points_.size() / out1.size();
  point_vec::iterator b = points_.begin(), e;

  for(size_t i = 0; i < out1.size(); ++i) {
    e = (i != out1.size() - 1) ? b + chunk : points_.end();
    out1[i]->send(point_range(b, e));
    b += chunk;
  }
  distribute_means();
}

// Iterate until there is no change in the means.
size_t producer::iterate()
{
  size_t niter = 0;
  bool changed;
  point p;

  do {
    ++niter;
    std::vector<point> new_means(means_.size(), ZEROPOINT);
    for(size_t i = 0; i < in.size(); ++i) {
      for(size_t j = 0; j < new_means.size(); ++j) {
        if(!in[i]->recv(p))
          abort();
        for(int k = 0; k < DIM; ++k)
          new_means[j].v[k] += p.v[k];
        new_means[j].c += p.c;
      }
    }

    changed = false;
    for(size_t i = 0; i < new_means.size(); ++i)
      for(int k = 0; k < DIM; k++)
        if((new_means[i].v[k] /= new_means[i].c) != means_[i].v[k])
          changed = true;

    if(changed) {
      means_.swap(new_means);
      distribute_means();
    }
  } while(changed);
```

```
  return niter;
}

void producer::distribute_means()
{
  for(size_t i = 0; i < out2.size(); ++i)
    for(size_t j = 0; j < means_.size(); ++j)
      out2[i]->send(means_[j]);
}

void producer::fill(point_vec &pv, size_t n)
{
  pv.reserve(n);
  while(n--) {
    point p;
    for(int i = 0; i < DIM; ++i)
      p.v[i] = rand() % GRIDSIZE;
    p.c = -1;
    pv.push_back(p);
  }
}

///////////////////////////////////////////////////////////////////////////////
// Computation of partial means

class kmeans : public perun::actor
{
  point_range     points_;
  std::vector<point>  cur_means_;

  static int square(int x)
  { return x*x; }

  static inline int distance(const point &p1, const point &p2);

protected:
  virtual void behavior();

public:
  perun::recv_port<point_range> in1;
  perun::recv_port<point>     in2;
  perun::send_port<point>     out;

  kmeans(size_t nmeans) : cur_means_(nmeans),
    in1(this), in2(this), out(this)
  { }
};

inline int kmeans::distance(const point &p1, const point &p2)
```

```
{
  int sum = 0;
  for(int k = 0; k < DIM; ++k) {
    int d = p1.v[k] - p2.v[k];
    sum += d*d;
  }
  return sum;
}

void kmeans::behavior()
{
  using boost::lambda::bind;
  using boost::lambda::_1;
  using boost::lambda::_2;

  if(!in1.recv(points_))
    abort();

  while(1) {
    point_vec partial_sums(cur_means_.size(), ZEROPOINT);

    for(size_t i = 0; i < cur_means_.size(); ++i)
      if(!in2.recv(cur_means_[i]))
        goto exit;

    for(point_vec::iterator it = get<0>(points_); it != get<1>(points_); ++it) {
      int c = std::min_element(cur_means_.begin(), cur_means_.end(),
        bind(&distance, *it, _1) < bind(&distance, *it, _2))
        - cur_means_.begin();
      it->c = c;
      for(int k = 0; k < DIM; ++k)
        partial_sums[c].v[k] += it->v[k];
      ++partial_sums[c].c;
    }

    for(size_t i = 0; i < partial_sums.size(); ++i)
      out.send(partial_sums[i]);
  }
exit:
  return;
}

//////////////////////////////////////////////////////////////////////////////
// DRIVER

struct params
{
  size_t  nworkers;
  size_t  npoints;
```

```
  size_t  nmeans;
};

static void bootstrap(params *p)
{
  producer *pprod = new producer(p->nworkers, p->npoints, p->nmeans);
  perun::RTE::spawn(pprod);

  for(size_t i = 0; i < p->nworkers; ++i) {
    kmeans *pk = new kmeans(p->nmeans);
    perun::RTE::connect(*pprod->out1[i], pk->in1, 128);
    perun::RTE::connect(*pprod->out2[i], pk->in2, 128);
    perun::RTE::connect(pk->out, *pprod->in[i]);
    perun::RTE::spawn(pk);
  }
}

int main(int argc, char **argv)
{
  params p = { 0, 100000, 100 };
  unsigned ncpus;

  if((argc != 5) && (argc != 3)) {
    cerr << "USAGE: " << argv[0] << " NCPUS NWORKERS "
      << "[NPOINTS=" << p.npoints << "] "
      << "[NMEANS=" << p.nmeans << "]\n";
    return 1;
  }

  ncpus = atoi(argv[1]);
  p.nworkers = atoi(argv[2]);
  if(argc == 5) {
    p.npoints = atoi(argv[3]);
    p.nmeans  = atoi(argv[4]);
  }

  perun::RTE::start(ncpus, boost::bind(&bootstrap, &p));
  perun::RTE::finish();
  return 0;
}
```