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

Paul B. Beskow*, Håvard Espeland*, Håkon K. Stensland*, Preben N. Olsen*,
Ståle Kristoffersen*, Carsten Griwodz, Pål Halvorsen

*Student author

Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{paulbb, haavares, haakonks, prebenno, staalebk, griff, paalh}@ifi.uio.no

1. INTRODUCTION

As the number of multimedia services grows, so does the computational demands on multimedia data processing. New multi-core hardware architectures provide the required resources, however, parallel, distributed applications are much harder to write than sequential programs. Large processing frameworks like Google's MapReduce [1] and Microsoft's Dryad [2] are steps in the right direction, but they are targeted towards batch processing. As such, we present *P2G*, which is a framework designed to integrate concepts from modern batch processing frameworks into the world of real-time multimedia processing. With P2G we seek to scale transparently with the available resources (following the cloud computing paradigm) and to support heterogeneous computing resources, such as GPU processing cores. The idea is to encourage the application developer to express as fine a granularity as possible along two axes, data and functional parallelism, where many of the existing systems sacrifice flexibility in one axis to accommodate for the other, e.g., MapReduce has no flexibility in the functional domain, but allows for fine-grained parallelism in the data domain. In P2G, functional blocks are formulated as kernels that operate on slices of multi-dimensional fields. As such, the fields, used to storing of the multimedia data, are used to express data decomposition. The write-once semantics of the fields provide the needed boundaries and barriers for functional decomposition to exist in our run-time and ensures deterministic output. P2G has intrinsic support for deadlines, and the compiler and run-time analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring. At the time of writing, we have implemented a prototype of a P2G execution node, with MJPEG as a primary workload, which we are working on optimizing as it at the moment is limited by our implementation of the dependency checker. Once the run-time is optimized and can scale properly with the number of cores available, we will also have results to present.

2. ARCHITECTURE

As shown in figure 1, P2G consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (i.e., multi-core, GPU, etc) to the master node, which combines this information to form a global topology of available resources. As such, the global

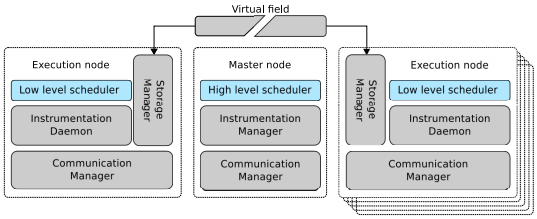
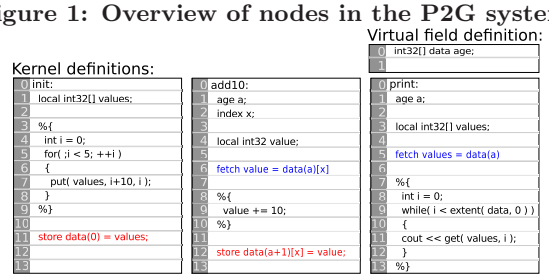
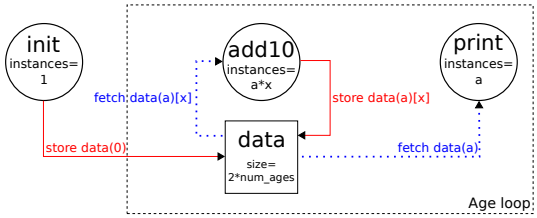


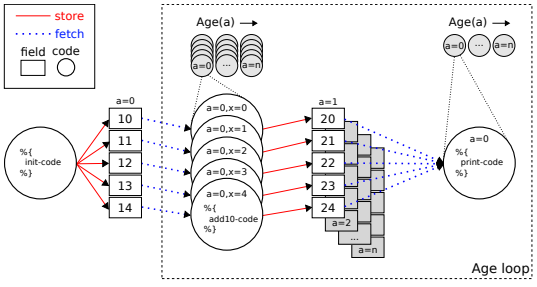
Figure 1: Overview of nodes in the P2G system.



(a) Kernel and virtual field definitions



(b) Implicit dependency graph



(c) Kernel and field instances

Figure 2: P2G programming model

topology can change during run-time as execution nodes can be dynamically added and removed to accommodate for changes in the global load.

As the master node receives workloads, it use its high-

level scheduler to determine which execution nodes to delegate partial or complete parts of the workload to. This process can be achieved in a number of ways. However, as a workload in P2G forms an implicit dependency graph based on its store and fetch operations to virtual fields, the high-level scheduler can utilize graph partitioning algorithms, or similar, to map such an implicit dependency graph to the global topology. The utilization of available resources is thus maximized.

When an implicit graph is split across multiple execution nodes, communication is achieved through an event-based, distributed publish-subscribe model. For every input, these subscriptions are deterministically derived from the code and the high-level schedulers partitioning decisions. The subscriptions also make it possible to establish direct communication links between the interacting execution nodes.

P2G uses a low-level scheduler at each execution node to maximize the local scheduling decisions, i.e., the low-level scheduler can decide to combine functional and data decomposition to minimize overhead. During run-time the master node will collect statistics on resource usage from all execution nodes, which all run an instrumentation daemon to acquire this information. The master node can then combine this run-time instrumentation data with the implicit dependency graph derived from the source code and the global topology to make continuous refinements to the high-level scheduling decisions. As such, P2G relies on its combination of a high-level scheduler, low-level schedulers, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

As seen in figure 2, P2G provides a kernel language for the programmer to write their application in, which they do by writing isolated, sequential pieces of code called *kernels*. Kernels operate on slices of *fields* through *fetch* and *store* operations and have native code embedded within them. In the model we encourage the programmer to specify the inherent parallelism in their application in as fine a granularity as possible in the domains of functional and data decomposition, without needing to sacrifice the one for the other.

The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to *fetch* slices of data in as fine granularity as possible. The write-once semantics of the fields provide deterministic output, though not necessarily deterministic execution of individual kernels. Given write-once semantics, iteration is supported in P2G by introducing the concept of aging, as seen in figure 2(b), where storing and fetching to the same field position, at different ages, makes it possible to form loops. The write-once semantics also provide natural boundaries and barriers for functional decomposition, as the low-level scheduler can analyze the dependencies of a kernel instance to determine if it is ready for execution. Furthermore, the compiler and the run-time, can analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring.

Given a workload specified using the P2G kernel language, P2G is designed to compile the source code for a number of heterogeneous architectures, though it currently only does so for the x86 architecture. P2G can then distribute this workload across the resources available to it.

At the time of writing, P2G consists of what we call an execution node, which is capable of executing entire workloads on a single x86 multi-core node. As such, the high-level scheduler and distribution mechanisms are not yet implemented, though the work is well under way.

3. WORKLOAD

We have implemented a few simple workloads used in multimedia processing to test the prototype implementation, here we will focus on our Motion JPEG implementation.

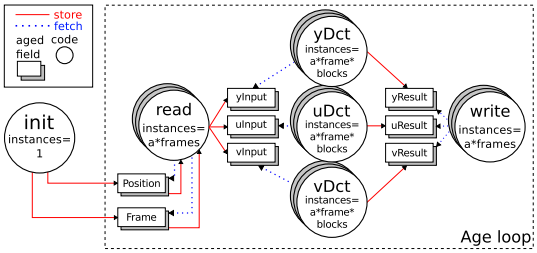


Figure 3: Overview of the P2G MJPEG encoding process

Motion JPEG is a sequence of separately compressed JPEG images. The MJPEG format provides several layers of parallelism, well suited for illustrating the potential of P2G. We focused on optimizing the discrete cosine transform (DCT) and quantization part, as this is the most compute-intensive part of the code. The *init* kernel starts the looping *read* kernel by storing to *position* and *frame's* first age (*age=0*). The *read* kernel fetches the most current age of these fields, utilizes the information, updates it, and stores it in the next age. This makes the *read* kernel dependent on itself and thus implicitly creates a loop. This loops ends when the kernel stops storing to the next age, e.g. at EOF. The YUV components can be processed independently, this property is exploited by creating three different kernels, *yDct*, *uDct*, and *vDct*. Each DCT kernel works on a single macro-block. Given a 352x288 resolution, this generates 1584 kernels of Y (luminance) data and 396 kernels of U and V (chroma) data. The *read* kernel stores the YUV data in three global fields, *yInput*, *uInput*, and *vInput*. From figure 3, we see that the respective DCT kernels are dependent on one of these fields.

4. POSTER & DEMO

In this poster/demo, we will explain and discuss the P2G ideas for multimedia processing with deadlines. Additionally, we accompany the poster with a demo of several well known multimedia workloads and show the entire application development and processing pipeline, i.e, the code in kernel language, the compilation with parallel code generation and the processing automatically distributing the multimedia load to the available processing cores.

5. REFERENCES

[1] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI* (2004), pp. 10–10.

[2] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys* (New York, NY, USA, 2007), ACM, pp. 59–72.