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

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

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

***Abstract*—The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multi-core hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counter-part. For this reason, Google and Microsoft implemented their respective processing frameworks MapReduce [10] and Dryad [19], as they allow the developer to think sequentially, yet benefit from parallel and distributed execution. An inherent limitation in the design of these *batch* processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs, or even pre-determined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution.**

With the Nornir runtime system for parallel programs [39], which is a Kahn Process Network implementation, we addressed and solved several of these limitations. However, it is more difficult to use than other frameworks due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, called *P2G*, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches and deadlines, and provides both data- and task-parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We have implemented an (interchangeable) P2G *kernel language* to ease development. In this paper, we present a proof of concept implementation of a P2G execution node and some experimental examples using complex workloads like Motion JPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

I. Introduction

Live, interactive multimedia services are steadily growing in volume. Interactively refined video search, dynamic participation in video conferencing systems and user-controlled views in live media transmissions are a few examples of features that future consumers will expect when they consume multimedia content. New usage patterns, such as extracting features in pictures to identify objects, calculation of 3D depth information from camera arrays, or generating free-view videos from multiple camera sources in real-time, add further magnitudes of processing requirements to already computationally intensive tasks like traditional video encoding. This fact is further exacerbated by the advent of high-definition videos.

Many-core systems, such as graphic processor units (GPUs), digital signal processors (DSPs) and large scale distributed systems in general, provide the required processing power, but taking advantage of the parallel computational capacity of such hardware is much more complex than single-core solutions. In addition, heterogeneous hardware requires individual adaptation of the code, and often involve domain specific knowledge. All this places additional burdens on the application developer. As a consequence, several frameworks have emerged that aim at making distributed application development and processing easier, such as Google's MapReduce [10] and Microsoft's Dryad [19]. These frameworks are limited by their design for batch processing of large amounts of data, with few dependencies across a large cluster of machines. Modifications and enhancements that address bottlenecks [8] together with support for new types of workloads and additional hardware exist [9], [16], [31]. It is also worth mentioning that new languages for current batch frameworks have been proposed [29], [30]. However, the development and processing of distributed multimedia applications is inherently more difficult. Multimedia applications also have stricter requirements for flexibility. Support for iterations is essential, and knowledge of deadlines is often imperative. The traditional batch processing frameworks do not support this.

In our Nornir runtime system for parallel processing [39], we addressed many of the shortcomings of the batch processing frameworks. Nornir is based on the idea of Kahn Process Networks (KPN). Compared to MapReduce-like approaches, Nornir adds support for arbitrary processing graphs, deterministic execution, etc. However, KPNs are designed with some unrealistic assumptions (like unlimited queue sizes), and the Nornir programming model is much more complex than that of frameworks like MapReduce and Dryad. It demands that the application developer establishes communication channels manually to form the dependency graph.

In this paper, we expand on our visions and present our initial ideas of *P2G*. It is a completely new framework for distributed real-time multimedia processing. P2G is designed

to work on continuous flows of data, such as live video streams, while still maintaining the ability to support batch workloads. We discuss the initial ideas and present a proof-of-concept prototype[1] running on x86 multi-core machines. We present experimental results using concrete multimedia examples. Our main conclusion is that the P2G approach is a step in the right direction for development and execution of complex parallel workloads.

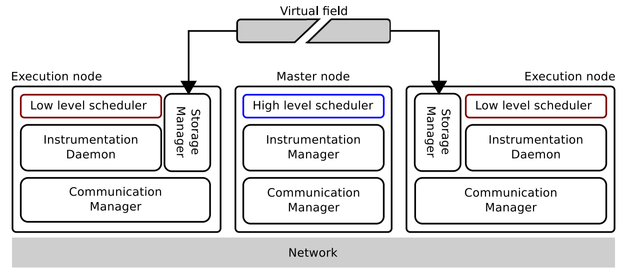

Figure 1. Overview of nodes in the P2G system.

II. Related Work

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

For example, several solutions have emerged for simplifying distributed processing of large quantities of data. We have already mentioned Google's MapReduce [10] and Microsoft's Dryad [19]. In addition, you have IBM's System S and accompanying programming language SPADE [13]. Yahoo have also implemented a programming language with their PigLatin language [29], other notable mentions for increased language support is Cosmos [26], Scope [6], CIEL [25], SNAPPLE [40] and DryadLINQ [41]. The high-level languages provide easy abstractions for the developers in an environment where mistakes are hard to correct.

Dryad, Cosmos and System S have many properties in common. They all use directed graphs to model computations and execute them on a cluster. System S also supports cycles in graphs, while Dryad supports non-deterministic constructs. However, not much is known about these systems, since no open implementations are freely available. MapReduce on the other hand has become one of the most cited paradigms for expressing parallel computations. While Dryad and System S use a task parallel model, MapReduce uses a data-parallel model based on keys and values. There are several implementations of MapReduce for clusters [1], multi-core [31], the Cell BE architecture [9], and also for GPUs [16]. Map-Reduce-Merge [8] adds a merge step to process data relationships among heterogeneous data sets efficiently, operations not directly supported by the original MapReduce model. In Oivos [35], the same issues are addressed, but in addition, this system provides a more expressive, declarative programming model. Finally, reducing the layering overhead of software running on top of MapReduce is the goal of Cogset [36] where the processing architecture is changed to increase performance.

An inherent limitation in MapReduce, Dryad and Cosmos is their inability to model iterative algorithms. In addition, the rigid MapReduce semantics do not map well to all types of problems [8], which may lead to unnaturally expressed solutions and decreased performance [38]. The limited support for iterative algorithms has been mitigated in HaLoop [5], a fork of Hadoop optimized for batch processing of iterative algorithms where data is kept local for future iterations of the MR steps. However, the programming model of MapReduce is designed for batch processing huge datasets, and not well suited for multimedia algorithms. Finally, Google's patent on MapReduce [11] may prompt commercial actors to look for an alternative framework.

KPN-based frameworks are one such alternative. KPNs support arbitrary communication graphs with cycles and are deterministic. However, in practice, very few general-purpose KPN runtime implementations exist. Known implementations include the Sesame project [34], the process network framework [28], YAPI [22] and our own Nornir [39]. These frameworks have several benefits, but for application developers, the KPN model has some challenges, particularly in a distributed scenario. To mention some issues, a distributed version of a KPN implementation requires a distributed deadlock detection and a developer must specify communication channels between the processes manually.

An alternative framework based on a process network paradigm is StreamIt [15], which comprises a language and a runtime system for simplifying the implementation of stream programs described by a graph that consists of computational blocks (filters) with a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous data-flow process network [23]. The compiler produces code that can make use of multiple machines or CPUs, whose number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

The processing and development of distributed multimedia applications is inherently more difficult than traditional sequential batch applications. Multimedia applications have strict requirements and knowledge of deadlines is necessary, especially in a live scenario. For multimedia applications that enable live communication, iterative processing is essential. Also, elastic scaling with the available resources becomes imperative when the workload, requirements or machine resouces change. Thus, all of the existing frameworks have some short-comings that are difficult to address, and the traditional batch processing frameworks simply come up short in our multimedia scenario. Next, inspired by the strengths of the different approach, we present our ideas for a new framework for distributed real-time multimedia processing.

[1]The P2G source code and workload examples are available for download from http://www.p2gproject.org/.

III. Basic idea

The idea of P2G was born out of the observation that most distributed processing framework lack support for real-time multimedia workloads, and that data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they can severely limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but restrict runtime processing to the domain of data parallelism [12]. Functional languages such as Erlang [3] and Haskell [18] and the event-based SDL [21], map well to task parallelism. Programs are expressed as communicating processes either through message passing or event distribution, which makes it difficult to express data parallelism without specifying a fixed number of communication channels.

In our multimedia scenario, Nornir improves on many of the shortcomings of the traditional batch processing frameworks, like MapReduce and Dryad. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs support also arbitrary communication graphs (with cycles/iterations), while frameworks like MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs). However, Nornir is task-parallel, and data-parallelism must be explicitly added by the programmer. Furthermore, as a distributed, multi-machine processing framework, Nornir still has some challenges. For example, the message-passing communication channels, having exactly one sender and one receiver, are modeled as infinite FIFO queues. In real-life distributed implementations, however, queue length is limited by available memory. A distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually. This requires the developer to think differently than for other distributed frameworks.

With P2G, we build on the knowledge gained from developing Nornir and address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature for processing multimedia workloads includes automatic combined task and data parallelism. Intra-frame prediction in H.264 AVC, for example, introduces many dependencies between sub-blocks of a frame, and together with other overlapping processing stages, these operations have a high potential for benefiting from both types of parallelism. We demonstrated the potential in earlier work with Nornir, whose deterministic nature showed great parallelization potential in processing arbitrary dependency graphs.

Multimedia algorithms being iterative by nature exhibit many pipeline parallel opportunities. Exploiting them are hard because intrinsic knowledge of fine-grained dependences are required, and structuring programs in such a way that pipeline parallelism can be used is difficult. Thies et al. [33] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses assuming that the behaviour is stable. They evaluated their system on multimedia algorithms and gained significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, and this enable the runtime to automatically detect and take full advantage of all parallel opportunities without manual intervention.

A major source of non-determinism in other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. The source of this non-deterministic behavior can be removed by adopting strict write-once semantics for writing to memory [4]. Languages that take advantage of the concept of single assignment include Erlang [3] and Haskell [18]. It enables schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once-semantics are well-suited for a scheduler's dependency analysis, it is not straight-forward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multi-dimensional arrays of data (e.g., pixels in a picture) and provide frequently very intuitive data partitioning opportunities (e.g., 8x8-pixel macro-blocks of a picture). Prominent examples are the computation-heavy MPEG-4 AVC encoding [20] and SIFT [24] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without loosing write-once-semantics.

Flexible partitioning requires the processing of clearly distinct data units without side-effects. The idea adopted for P2G is to use *kernels* as in stream processing [15], [27]. Such a kernel is written once and describes the transformation of multi-dimensional fields of data. Where such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which can analyze dependencies and guarantee fully deterministic output independent of order due to the write-once semantics of fields.

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

- The use of *multi-dimensional fields* as the central concept for storing data in P2G to achieve straight-forward implementations of complex multimedia algorithms.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *write-once semantics* to such fields to achieve deterministic behavior.

- The use of *runtime dependency analysis* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

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

P2G is designed to be language independent, however, we have defined a C-like language that captures many of P2G's central concepts. As such, the P2G language is inspired by many existing languages. In fact, Cray's Chapel [7] language antedates many of P2G's features in a more complete manner. P2G adds, however, write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements that formulate data dependencies between implicitly instantiated kernels, which are (currently) written in C/C++.

The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations. Furthermore, P2G supports deadlines, which allows scheduling decisions such as termination, branching and the use of alternative code paths based on runtime observations.

In summary, we have opted for an idea that allows programmers to focus on data transformations in a sequential manner, while simultaneously providing enough information for dynamically adapting the data and task parallelization. As an end result of our considerations. P2G's fields look mostly like global multi-dimensional arrays in C, although their representation in memory may deviate, i.e., they need not be placed contiguously in the memory of a single node, and may even be distributed across multiple machines. Although this looks contrary to our message-based KPN approach used in Nornir, it maps well when slices of fields are interpreted as messages and the run-queues of worker threads as KPN channels. An obvious difference is that fields can be read as often as necessary.

IV. Architecture

As shown in figure 1, the P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (a graph of multi-core and single-core CPUs and GPUs, connected by various kinds of buses and other networks) to the master node, which combines this information into a global topology of available resources. As such, the global topology can change during runtime as execution nodes are dynamically added and removed to accommodate for changes in the global load.

To maximize throughput, P2G uses a two-level scheduling approach. On the master node, we have a high-level scheduler (HLS), and on the execution node(s), we use a low-level scheduler (LLS). The HLS can analyze a workloads

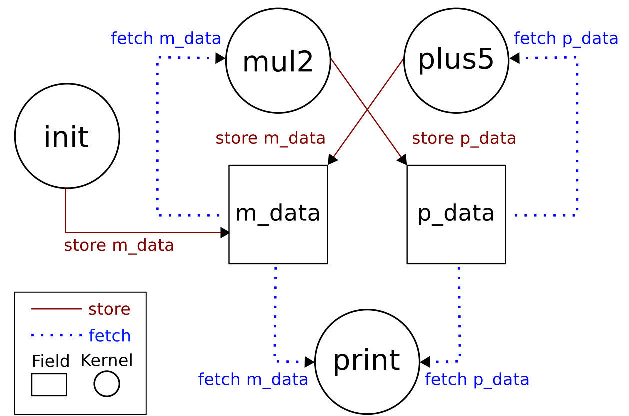

Figure 2. Intermediate implicit static dependency graph

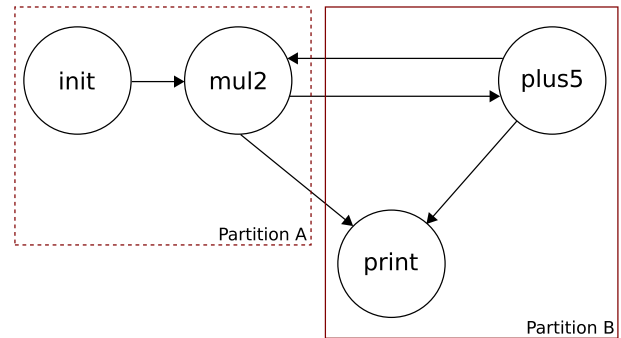

Figure 3. Final implicit static dependency graph

store and fetch statements, from which it can generate an intermediate implicit static dependency graph (see figure 2) where edges connecting two kernels through a field can be merged, circumventing the need for a vertex representing the field (as seen in figure 3). From the intermediate graph, the HLS can then derive a final implicit static dependency graph (see figure 3). The HLS can then use a graph partitioning [17] or search based [14] algorithm to partition the workload into a suitable number of components that can be distributed to, and run, on the resources available in the topology. Using instrumentation data collected from the nodes executing the workload the final graph can be weighted with this profiling data during runtime. The weighted final graph can then be repartitioned, with the intent of improving the throughput in the system, or accommodate for changes in the global load.

Given a partial workload (such as *partition A* from figure 3), an LLS at an execution node is responsible for maximizing local scheduling decisions. We discuss this further in section V, but figure 4 shows how the LLS can combine tasks and data to minimize overhead introduced by P2G, and take advantage of specialized hardware, such as GPUs.

This idea of using a two level scheduling approach is not new. It has also been considered by Roh et al. [32], where they have performed simulations on parallel scheduling decisions for instruction sets of a functional language. Simple workloads are mapped to various simulated architectures, using a "merge-

up" algorithm, which is equivalent to our LLS, and "merge-down" algorithm, which is equivalent to our HLS. These algorithms cluster instructions in such a way that parallelism is not limited, their conclusion is that utilizing a merge-down strategy often is better.

Data distribution, reporting, and other communication patterns is achieved in P2G through an event-based, distributed publish-subscribe model. Dependencies between components in a workload are deterministically derived from the code and the high-level schedulers partitioning decisions, and direct communication occurs.

As such, P2G relies on its combination of a HLS, LLS, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

V. Programming model

The programming model of P2G consists of two central concepts, the *implicit static dependency graph* (figures 2 and 3) and the *dynamically created directed acyclic dependency graph* (DC-DAG) (figure 4). We have also developed a *kernel language* (see figure 5), to make it easier to develop applications using the P2G programming model, though we consider this language to be interchangeable.

The example we use throughout this discussion consists of two primary kernels: *mul2* and *plus5*. These two kernels form a pipeline where *mul2* first multiples a value by 2 and stores this data, which *plus5* then fetches and increases by 5, *mul2* then fetches the data stored by *plus5*, and so on. The *print* kernel runs orthogonally to these two kernels and fetches and writes the data they have produced to *cout*. In combination, these three kernels form a cycle. The kernel *init* runs only once and writes some initial data for *mul2* to consume. The kernels operate on two 1-dimensional, 5 element fields. The print kernel writes *{10, 11, 12, 13, 14}*, *{20, 22, 24, 26, 28}* for the first *age* and *{25, 27, 29, 31, 33}*, *{50, 54, 58, 62, 66}* for the second, etc (as seen in figure 4). As such, the first *iteration* produces the data: *{10, 11, 12, 13, 14}*, *{20, 22, 24, 26, 28}* and *{25, 27, 29, 31, 33}*, and the second *iteration* produces the data: *{50, 54, 58, 62, 66}* and *{55, 59, 63, 67, 71}*, etc. Since there is no termination condition for this program it runs indefinitely.

A. Dependency graphs

The intermediate implicit static dependency graph (as seen in figure 2) is derived from the interaction between fields and kernel definitions, more precisely from the *fetch* and *store* statements of a kernel definition. This intermediate graph can be further refined by merging the edges of kernels linked through a field vertex, resulting in a final implicit static dependency graph, as depicted in figure 3. This final graph can serve as input to the HLS, which can use it to determine how best to partition the workload given a global topology. The graph can be further weighted using instrumentation data, to serve as input for repartitioning. It is important to note that these weighted graphs can serve as input to static offline analysis. For example, it could be used as input to a simulator to best determine how to initially configure a workload, given various global topology configurations.

During runtime, the intermediate implicit static dependency graph is expanded to form a dynamically created directed acyclic dependency graph, as seen in figure 4. This expansion from a cyclic graph to a directed acyclic graph occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iteration. We have chosen to call each such unrolled loop an *Age*. The LLS can then use the DC-DAG to combine tasks and data to reduce overhead introduced by P2G and to take advantage of specialized hardware, such as GPUs. It can then try different combinations of these low-level scheduling decisions to improve the throughput of the system.

We can see how this is accomplished in figure 4. When moving from *Age=1* to *Age=2*, we can see how the LLS has made a decision to reduce data parallelity. In P2G, kernels fetch slices of data, and initially *mul2* was defined to work on each single field entry in parallel, but in *Age=2*, the LLS has decreased the granularity of the fetch statement to encompass the entire field. It could also have split the field in two, leading to two kernel instances of *mul2*, working on disparate sets of the field.

Moving from *Age=2* to *Age=3*, we see how the LLS has made a decision to decrease the task parallelity. This is possible because *mul2* and *plus5* effectively form a pipeline, information that is available from the static graphs. By combining these two tasks, the individual store operations of the tasks are deferred until the data has been fully processed by each task. If the *print* kernel was not present, storing to the intermediate field *m_data* could be circumvented in its entirety.

Finally, moving from *Age=3* to *Age=4*, we can see how a decision to decrease both task and data parallelity has been taken. This renders this single kernel instance effectively instance into a classical *for-loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the data.

P2G makes runtime adjustments dynamically to both data and task parallelism based on the possibly oscillating resource availability and the reported performance monitoring.

B. Kernel language

From our experience with developing Nornir, we came to the realization that expressing workloads in a framework capable of supporting such complex graphs without a high-level language is a difficult task. We have therefore developed a *kernel language*. An implementation of a simple workload is outlined in figure 5, with a C++ equivalent listed in figure 6.

In the current version of our system, P2G is exposed to the developer through this *kernel language*. The language itself is not an integral part and can be replaced easily. However, it exposes several foundations of the P2G design. Most important are the kernel and field definitions, which describe the code and interaction patterns in P2G.

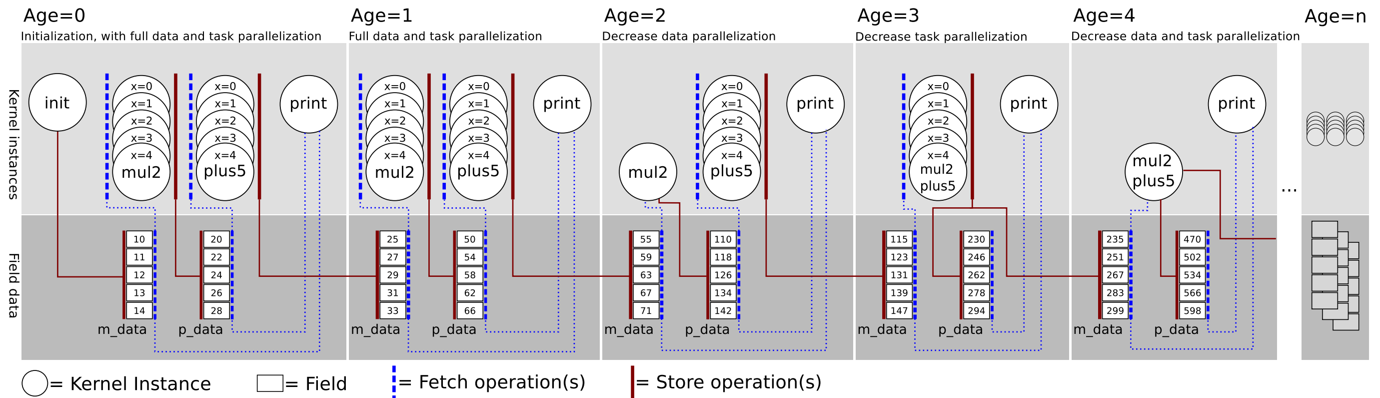

Figure 4. Dynamically created directed acyclic dependency graph (DC-DAG)

Field definitions:

```
0 int32[] m_data age;
1 int32[] p_data age;
```

Kernel definitions:

```
0  init:
1    local int32[] values;
2
3    %{
4      int i = 0;
5      for( ;i < 5; ++i )
6      {
7        put( values, i+10, i );
8      }
9    %}
10
11   store m_data(0) = values;
12
13
```

```
0  mul2:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = m_data(a)[x];
6
7    %{
8      value *= 2;
9    %}
10
11   store p_data(a)[x] = value;
12
13
```

```
0  plus5:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = p_data(a)[x];
6
7    %{
8      value += 5;
9    %}
10
11   store m_data(a+1)[x] = value;
12
13
14
15
```

```
0  print:
1    age a;
2    local int32[] m, p;
3
4    fetch m = m_data(a);
5    fetch p = p_data(a);
6
7    %{
8    for(int i=0; i < extent(m, 0);)
9      cout << get(m, i++) << " ";
10   cout << endl;
11
12   for(int i=0; i < extent(p, 0);)
13     cout << get(p, i++) << " ";
14   cout << endl;
15 %}
```

Figure 5. Kernel and field definitions

A kernel definition's primary purpose is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through the fetch and store statements. As such, a field serves as an interaction point for kernel definitions, as can be seen in figure 2.

An important aspect of multimedia workloads is the ability to express deadlines, where it does not make sense to encode a frame if the playback has moved past that point in the video-stream. Consequently, we have implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer: *timer t1*. This timer can then be polled, and updated, from within a kernel definition, for example *t1+100ms* or *t1 = now*. Given a condition based on a deadline such as *t1+100ms*, a timeout can occur and an alternate code-path can be executed. Such an alternate code-path is executed by storing to a different field then in the primary path, leading to new dependencies and new behavior. Currently, we have basic support for expressing deadlines in the kernel language, but the semantics of these expressions require refinement, as their implications can be considerable.

Fields in P2G have a number of properties, including a type and a dimensionality. Another property is, as mentioned above, *aging*, which allows kernels to be iterative while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times,

```cpp
void print( int *data, int num )
{
    for( int i = 0; i < num; ++i )
        std::cout << data[i] << "␣";
    std::cout << std::endl;
}
int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    while( true )
    {
        for( int i = 0; i < 5; ++i )
            p_data[i] = m_data[i] * 2;

        print( m_data, 5 );
        print( p_data, 5 );

        for( int i = 0; i < 5; ++i )
            m_data[i] = p_data[i] + 5;
    }
    return 0;
}
```

Figure 6. C++ equivalent of mul/sum example

as long as the age increases for each store operation (as seen in figure 4). In essence, this adds a dimension to the field and makes it possible to accommodate iterative algorithms. Additionally, it is important to realize that fields are not connected to any single node, and can be fully localized or distributed across multiple execution nodes (as seen in figure 1).

In defining the interaction between kernels and fields, it is encouraged that the programmer expresses the finest possible granularity of kernel definitions, and, likewise, the most precise slices possible for the kernel within the field. This is encouraged because it provides the low-level scheduler more control over the granularity of task and data decomposition. Aided by instrumentation data, it can reduce scheduling overhead by combining several instances of a kernel that process different data, or several instances of different kernels that process data in sequence (as seen in figure 4). The scheduler makes its decisions based on the implicit static dependency graph and instrumentation data.

C. Runtime

Following from the previous discussions, we can extrapolate the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime, and the number of kernel instances executed in parallel for a given kernel definition depends on its fetch statements.

To clarify, a kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements of the kernel definition. For example, looking at figure 4 and 5, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x* fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum possible *x* kernel instances simultaneously per age, giving *a*x* *mul2* kernel instances. Though, as we have seen, this number can be decreased by the scheduler making *mul2* work over larger slices of data from *m_data*.

With P2G we support implicit resizing of fields, this can be witnessed by looking at the kernel definition of *print* in figure 5. Initially, the extents of *m_data* and *p_data* are not defined, as such, with each iteration of the *for*-loop in *init* the local field *values* is resized locally, leading to a resize of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields impacted by this resize, such as *p_data*. Following the discussion from the previous paragraph, such an implicit resize can lead to additional kernel instances being dispatched.

It is worth noting that a kernel instance is only dispatched when all its dependencies are fulfilled, i.e., that the data it fetches has been stored to the respective fields and elements. Looking at figure 4 and 5 again, we can see that *mul2* stores its result to *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, this means that the kernel instance *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. In our system, each kernel instance is only dispatched once, due to our write-once semantics. To summarize, the *print* kernel instance working on *age=0* becomes runnable when all the elements of *m_data* and *p_data* for *age=0* have been stored. Once it has become runnable, it is dispatched and runs only once.

VI. Prototype Implementation

To verify the feasibility of the P2G framework presented in this paper, we have implemented a prototype version. The prototype consists of a compiler for the kernel language and a runtime that can execute P2G programs on multi-core linux machines.

A. Compiler

Programs written for the P2G system are designed to be platform independent and feature native blocks of code written in C or C++. Heterogeneous systems are specifically targeted, but many of these require a custom compiler for the native blocks, such as nVIDIA's nvcc compiler for the CUDA system and IBM's XL compiler for the Cell Broadband Engine. We decided to compile P2G programs into C++ files, which can be further compiled and linked with native code blocks, instead of generating binaries directly. This approach gives us less control of the resulting object code, but we gain the flexibility and sophisticated optimization of the native compilers, resulting in a lightweight P2G compiler. The P2G compiler works also as a compiler driver for the native compiler and produces complete binaries for programs that run directly on the target system.

B. Runtime

The runtime prototype implements the basic features of a P2G execution node, including multi-dimensional field support, implicit resizing of fields, instrumentation and parallel execution of kernel instances on multiple processors using the implicit dependency graph formed by kernel definitions. However, at the time of writing, the prototype runtime does not yet have a full implementation of deadline expressions, this is because the semantics of the kernel language support for this feature is not fully defined yet.

The prototype targets a node with multiple processors. It is designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which may require resize operations. A kernel subscribes to events related to fields that it depends on, i.e., fields referenced to by the kernels *fetch* statements. When receiving such a storage event, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement, and puts these in a per-kernel ready queue. This means that the ready queues contain always the maximum number of parallel instances that can be executed at any time, only limited by unfulfilled data dependencies.

The low-level scheduler consists of a dependency analyzer and kernel instance dispatcher. Using the implicit dependency graph, the dependency analyzer adds new kernel instances to a ready queue, which later can be processed by the worker

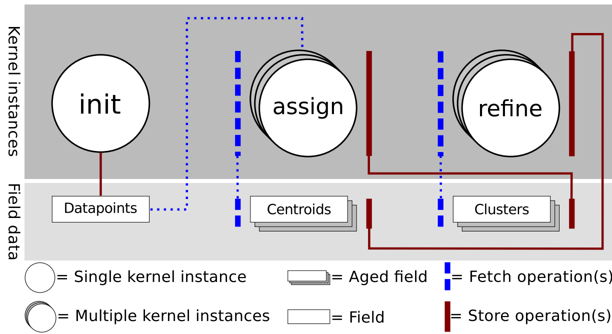

Figure 7. Overview of the K-means clustering algorithm

threads. Dependencies are analyzed in a dedicated thread which handles events emitted from running kernel instances that notifies on *store* and *resize* operations performed on fields. Kernel instances are executed by a worker thread dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements or by groups of kernels that satisfy their own dependencies in aging cycles, such as the *mul2* and *plus5* kernel in figure 5.

The runtime is written in C++ and uses the blitz++ [37] library for high-performance multi-dimensional arrays. The source code for the P2G compiler and runtime can be downloaded from http://www.p2gproject.org/.

VII. Workloads

We have implemented a few workloads commonly used in multimedia processing to test the prototype implementation. The P2G kernel language is able to expose both the data and task parallelism of the programs to the P2G system, so the runtime is able to adapt execution of the programs to suit the target architecture.

A. K-means clustering

K-means clustering is an iterative algorithm for cluster analysis which aims to partition n datapoints into k clusters in which each datapoint belongs to the cluster with the nearest mean. As shown in figure 7, the P2G k-means implementation consists of an *init* kernel, which generates n datapoints and *stores* them to the datapoints field. Then, it selects k of these datapoints randomly, as the initial means, and *stores* them to the centroids field. Next, the *assign* kernel *fetches* a slice of data, a single datapoint per *kernel instance*, the last calculated centroids, and *stores* this datapoint to the cluster of the closest centroids using the euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* form a loop which gradually leads to a convergence in centroids, at which point the k-means algorithm has completed.

B. Motion JPEG

Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images. The MJPEG

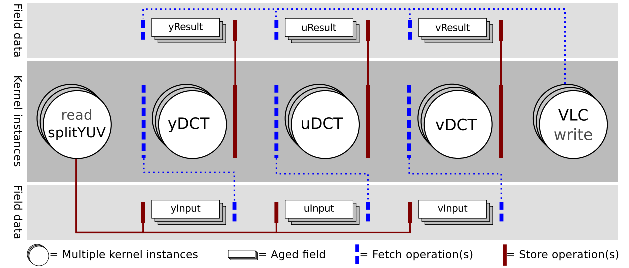

Figure 8. Overview of the MJPEG encoding process

4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table I
Overview of test machines

format provides many layers of parallelism, well suited for illustrating the potential of the framework. We focused on optimizing the discrete cosine transform (DCT) and quantization part as this is the most compute-intensive part of the codec.

The *read + splitYUV* kernel reads the input video in YUV-format and stores the data in three global fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing to the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 8, we see that the respective DCT kernels are dependent on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*. Finally, the *VLC + write* kernel store the MJPEG bit-stream to disk.

VIII. Evaluation

We have run tests with the workloads Motion JPEG and K-means (described in section VII). Each test was run on a *4-way Core i7* and an *8-way Opteron* (see table I for hardware specifications) ranging from *1* worker thread to *8* worker threads with *10* iterations per worker thread count. The results of these tests are reported in the figures 10 and 9, which show the mean running time in *seconds* for each machine for a given thread count with standard deviation reported as error-bars.

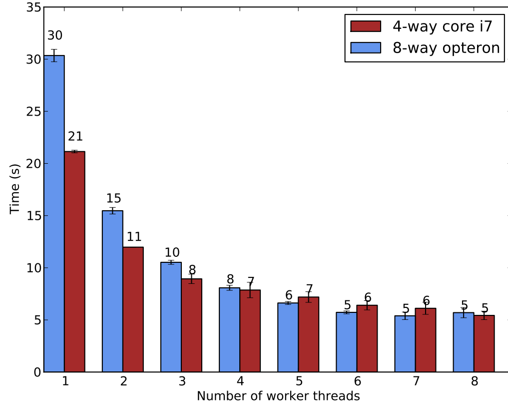

Figure 9. Workload execution time for Motion JPEG

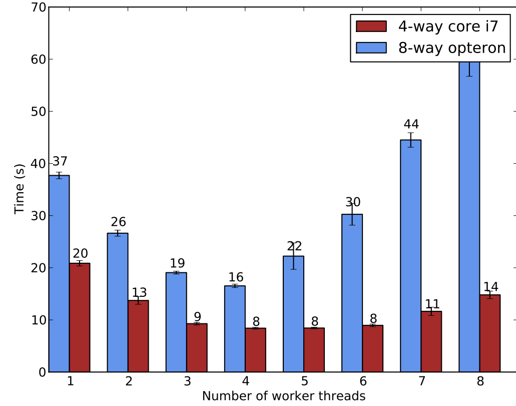

Figure 10. Workload execution time for K-means

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1641.57 μs
yDCT	80784	3.07 μs	170.30 μs
uDCT	20196	3.14 μs	170.24 μs
vDCT	20196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2160.71 μs

Table II
MICRO-BENCHMARK OF MJPEG ENCODING IN P2G

In addition, we have performed micro-benchmarks for each workload, summarized in the tables II and III. The benchmarks summarize the number of kernel instances dispatched per kernel definition, dispatch overhead and time spent in kernel code.

A. *Motion JPEG*

The Motion JPEG workload is run on the standard test sequence *Foreman* encoded in *CIF* resolution. We limited the workload to process *50* frames of video.

As we can observe from figure 9, P2G is able to scale close to linearly with the resources it has available. In P2G, the dependency analyzer of the LLS runs in a dedicated thread. This affects the running time when moving from *7* to *8* worker threads. Where the eighth thread shares resources with the dependency analyzer. To compare, the standalone single threaded MJPEG encoder on which the P2G version is based upon has a running time of *30* seconds on the Opteron machine and *19* seconds on the Core i7 machine. Note that both the standalone and P2G versions of the MJPEG encoder use a naive DCT calculation, there are versions of DCT that can significantly improve performance, such as FastDCT [2].

From table II, we can see that time spent in kernel code is considerably higher compared to the dispatch overhead for the kernel definitions. The dispatch time includes allocation or reallocation of fields as part of the timing operation. As a result, *init* and *read/splitYUV* have a considerably higher dispatch time then the **DCT* operations.

We can also see that the majority of CPU-time is spent in the kernel instances of *yDCT, uDCT* and *vDCT*, which is the computationally intensive part of the workload. This indicates that decreasing data and task granularity, as discussed in section V-A, has little impact on the throughput of the system. This is because the majority of time is already spent in kernel code.

Note that even though there are *51* instances of the *read-/write* kernel definitions, only *50* frames are encoded, because the last instance reaches the end of the video stream.

B. *K-means*

The K-means workload is run with K=*100* using a randomly generated data set containing *2000* datapoints. The K-means algorithm is not run until convergence, but with *10* iterations. If we do not define this break-point it is undefined when the algorithm converges, and as such, we have introduced this condition to ensure that we get a relatively stable running time for each run.

As seen in figure 10, the K-means workload scales to *4* worker threads. After this, the running time increases with the number of worker threads. This can be explained by the fine granularity of the *assign* kernel definition, as witnessed when comparing the dispatch time to the time spent in kernel code. This leads to the serial dependency analyzer becoming a bottle-neck in the system. As discussed in section V-A, this condition could be alleviated by decreasing the granularity of data-parallelism, in effect leading to each kernel instance of *assign* working on larger slices of data. By doing so, we would increase the ratio of time spent in kernel code compared to dispatch time and reduce the workload of the dependency analyzer. The reduction in work for the dependency analyzer is a result of the lower number of kernel instances being run.

The two different test machines behave somewhat differently in that the Opteron suffers more than the Core i7 when the dependency analyzer saturates a core. The Core i7 is able to increase the frequency of a single core to mitigate serial bottlenecks, and we think this is why the Core i7 suffers less when we meet the limitations dictated by Amdahl's law.

The considerable time *init* spends in kernel code is because it generates the data set.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9829.00 μs
assign	2024251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table III
MICRO-BENCHMARK OF K-MEANS IN P2G

C. Summary

We have shown that our prototype implementation of an execution node is able to scale with the available resources, as seen in figure 9 and 10. Our initial results indicate that the functionality of decreasing the granularity of task and data parallelity, as discussed in section V-A, is important to ensure full resource utilization.

IX. DISCUSSION

Even though support for deadlines is not yet fully implemented in the P2G runtime, the concept of deadlines formed an integral part of our design goal. The intention behind deadlines is to accommodate for live multimedia workloads, where real-time requirements are mission essential. Varying conditions over time, both in the workload and topology, may effect scheduling decisions: such as termination, branching and the use of alternative code paths based on runtime observations. This is similar to SDL, but unlike contemporary high performance languages.

In P2G, we encourage the programmer to describe the workload in as fine granularity as possible, both in the functional and data decomposition domains. The low-level scheduler has an understanding of both decomposition domains and deadlines. Given this information, the low-level scheduler can minimize overhead by combining functional components and slices of data by adapting to its available resources, be it local cores, or even GPU execution units.

Write-once semantics on fields incurs a large penalty if implemented naively, both in terms of memory usage and data cache misses. However, as the fields are virtual and do not even have to reside in continuous memory, the compiler and runtime are free to optimize field usage. This includes re-using buffers for increased cache locality when old ages are no longer referenced, and garbage collecting old ages. The explicit programming model of P2G allows the system to anticipate what data is needed in the future, which can be used for further optimizations.

Given the complexity of multimedia workloads and the (potentially) heterogeneous resources available in a modern topology, and in many cases, no knowledge of the underlying capabilities of the resources (which is common in modern cloud services), mapping these complex multimedia workloads manually to the available resources becomes an increasingly difficult task, and at some point, even impossible. This is particularly the case where resource availability fluctuates, such as in modern virtual machine parks. With batch processing, where the workloads frequently are not associated with some intrinsic deadline, this task is solved, with frameworks such as MapReduce and Dryad. However, for processing continuous streams such as iterative multimedia algorithms in an elastic manner requires new frameworks; P2G is a step in that direction.

X. CONCLUSION

With P2G, we have proposed a new flexible framework for automatic parallel, real-time processing of multimedia workloads. We encourage the programmer to specify parallelism in as fine a granularity as possible along the axes of data and task decomposition. Using our kernel language this decomposition is expressed through kernel definitions and fetch and store statements on fields. This language is independent from the P2G runtime and can easily be replaced. Given a workload defined in our kernel language it is compiled for execution in P2G. This workload can then be partitioned by the high-level scheduler of a P2G master node, which then distributes partitions to P2G execution nodes which runs the tasks locally. Execution nodes can consist of heterogeneous resources. A low-level scheduler at the execution nodes then adapted the partial (or full) workload to run optimally using resources at hand. Feedback from the instrumentation daemon at the execution node can lead to repartitioning of the workload (a task performed by the high-level scheduler). The aim is to bring the ease of batch-processing frameworks to multimedia workloads.

In this paper we have presented an execution node capable of running on a multi-way architecture. This results from our experiments running on this prototype show the potential of our ideas. However, there still remains a number of vectors for optimization. In the low-level scheduler we have identified that combining task and data to minimize overhead introduced by P2G is a first reasonable modification. Additionally, completing the implementation of a fully distributed version is in the pipeline. Also, writing workloads for heterogeneous processing cores like GPUs and non-cache coherent architectures like Intel's SCC is a further consideration. Currently, we are investigating appropriate mechanisms for both high- and low-level scheduling, garbage collection, fat binaries, resource profiling and monitoring, and efficient migration of tasks.

While a number of optimizations remain, we have determined that P2G is feasible, through the implementation of this execution node, and the successful implementation of multimedia workloads, such as Motion JPEG and k-means. With these workloads we have shown that it is possible to express multimedia workloads in the kernel language and we have implemented a prototype of an execution node in the P2G framework that is able to execute kernels and scales with the available resources.

REFERENCES

[1] Apache. Hadoop, Accessed July 2010. http://hadoop.apache.org.
[2] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of IEICE*, E71(11), 1988.
[3] J. Armstrong. A history of Erlang. In *Proc. of ACM HOTL III*, pages 6:1–6:26, 2007.
[4] R. Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, 1989.

[5] Y. Blu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

[6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[8] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[9] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

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

[11] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. *US Patent Application*, (US 7650331), 2010.

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

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[14] F. Glover. Tabu search, Part I1. *ORSA journal on Computing*, 2(1):4–32, 1990.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of PACT*, pages 260–269, New York, NY, USA, 2008. ACM.

[17] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing* 1. *Parallel Computing*, 26(12):1519–1534, 2000.

[18] P. H. J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of ACM HOTL III*, pages 12:1–12:55, 2007.

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

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

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

[22] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf. Yapi: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC'2000*, pages 402–405. ACM Press, 2000.

[23] T. Lee, E.A.; Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

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

[25] D. Murray, M. Schwarzkopf, and C. Smowton. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[26] C. Nicolaou. An architecture for real-time multimedia communication systems. *Selected Areas in Communications, IEEE Journal on*, 8(3):391–400, 1990.

[27] Nvidia. Nvidia cuda programming guide 3.2, Aug. 2010.

[28] A. G. Olson and B. L. Evans. Deadlock detection for distributed process networks. In *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP*, pages 73–76, 2006.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[30] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of IEEE HPCA*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[32] L. Roh, W. A. Najjar, and A. P. W. Böhm. Generation and quantitative evaluation of dataflow clusters. In *ACM FPCA: Functional Programming Languages and Computer Architecture*, New York, NY, USA, 1993. ACM.

[33] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[34] M. Thompson and A. Pimentel. Towards multi-application workload modeling in sesame for system-level design space exploration. In S. Vassiliadis, M. Berekovic, and T. Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222–232. Springer Berlin / Heidelberg, 2007.

[35] S. V. Valvåg and D. Johansen. Oivos: Simple and efficient distributed data processing. In *Proc. of IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 113–122, 2008.

[36] S. V. Valvåg and D. Johansen. Cogset: A unified engine for reliable storage and parallel processing. In *Proc. of IFIP International Conference on Network and Parallel Computing Workshops (NPC)*, pages 174–181, 2009.

[37] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '98, pages 223–230, London, UK, 1998. Springer-Verlag.

[38] Ẑ. Vrba, P. Halvorsen, C. Griwodz, and P. Beskow. Kahn process networks are a flexible alternative to mapreduce. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:154–162, 2009.

[39] Ẑ. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen. The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce. *Journal of Sumpercomputing*, 27(1), 2010.

[40] D. Waddington, C. Tian, and K. Sivaramakrishnan. Scalable lightweight task management for mimd processors, 2011.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. Dryadlinq: A system for general-puropose distributed data-parallel computing using a high-level language, 2008.