UNIVERSITY OF OSLO
Department of Informatics

Dynamic adaptation and distribution of binaries to heterogeneous architectures

Masteroppgave

Espen Angell Kristiansen

Abstract

Real time multimedia workloads require progressingly more processing power. Modern many-core architectures provide enough processing power to satisfy the requirements of many real time multimedia workloads. When even they are unable to satisfy processing power requirements, network-distribution can provide many workloads with even more computing power.

In this thesis, we present solutions that can be used to make it practical to use the processing power that networks of many-core architectures can provide. The research focus on solutions that can be included in our Parallel Processing Graphs (P2G) project.

We have developed the foundation for network distribution in P2G, and we have suggested a viable solution for execution of workloads on heterogeneous multi-core architectures.

Acknowledgements

I would like to express my gratitude to my supervisor, Håkon Stensland, for his excellent guidance, and my other supervisors, Pål Halvorsen and Carsten Griwodz, for valuable feedback during my study.

Thanks to everyone working on the P2G project. I would like to extends a special thanks to Paul Beskow and Ståle Kristoffersen for invaluable input while researching network distribution.

I also want to use this chance to say thank you to my friends, my family, and especially my girlfriend Hege, for supporting me through my studies.

Oslo, May 16, 2011
Espen Angell Kristiansen

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Background and motivation

Ever since the first integrated circuit was introduced in 1958, the strive for more processing power has been a persisting challenge. In 1965, Gordon Moore made his famous prediction, dubbed Moore's Law [1], that the number of transistors incorporated in a microchip will approximately double every 24 months. His prediction has held true until today [68]. However, Intel [68] predicts that alternative computing techniques will be required as soon as in 2020 to keep up with the computing power predicted by Moore's Law. Real time multimedia workloads, such as high quality real time video encoding and calculation of 3D depth information from camera arrays, require far more processing capacity than a single Central Processing Unit (CPU) core can achieve. Thus, we already require alternative computing techniques. Heterogeneous many-core architectures such as Graphics Processing Units (GPUs) and the Cell Broadband Engine (CBE) [69] can outperform CPUs by orders of magnitude [2–4] on many workloads.

A heterogeneous architecture is an architecture with multiple different processing units. This can be a computer with a CPU and a Graphics Processing Unit (GPU). We call such a computer a *heterogeneous computer*. Such a system is considered heterogeneous since the CPU and GPU are different processing units, each suitable for different tasks. More complex heterogeneous architectures are readily available. We can for example, build a computer with one or more CPUs, GPUs

from multiple different manufacturers, and a programmable network processor. This would provide us with a high amount of processing power and processing units optimized for many different workloads. However, making programs for such an architecture is complex and error prone.

A common way of handling workloads that require more processing power than a single computer can provide, is to distribute the workload over multiple computers in a network. Making programs for a single heterogeneous computer is complex and error prone, and distributing such workloads over a network adds another level of complexity. For this reason, Microsoft and Google implemented their respective processing frameworks MapReduce [5] and Dryad [6]. These frameworks allow developers to think sequentially, yet benefit from parallel and distributed execution. An inherent limitation of these frameworks is their inability to express arbitrarily complex graphs. Their dependency graphs are often limited to directed acyclic graphs, or pre-defined stages. Algorithms that depend on iterative execution, such as video encoding, require more complex dependency graphs.

The Nornir runtime system for parallel programs [7], address many of these limitations, however it is more difficult to use than other frameworks due to a more complex programming model. With the Parallel Processing Graphs (P2G) project, we seek to scale multimedia workloads efficiently and transparently with the available resources. P2G supports arbitrarily complex dependency graphs, with cycles, branches and deadlines. To reduce the complexity of programming applications for P2G compared to Nornir, we have adopted the use of *kernels*, as in stream processing [8] [70]. An executing kernel is called a *kernel instance*.

P2G is designed to be language independent. The kernel language is interchangeable, however our current kernel language captures the central concepts of P2G. Message passing and data parallelism is implicit, and developers can think in terms of sequential data transformations.

In a network-distributed version of P2G, we assume there will be an application, or multiple applications, running on each computer in the network. This application, which we call the *P2G runtime*, accepts jobs and communicates with the scheduler. P2G is discussed further in Chapter 2.

1.2 Problem Definition / Statement

At this time, P2G only supports execution on a single computer with a x86 multi-core CPU. The data shared between kernel instances is stored in memory accessible by the underlying framework and the kernels. In this thesis, we study methods for distribution of P2G kernel instances over all processing units in a network of computers. The goal of our research is to find methods enabling P2G applications to parallelize and distribute workloads, especially live multimedia workloads, in a network of many-core homogeneous and heterogeneous computers. Any method supporting heterogeneous computers also support homogeneous computers, therefore we focus on heterogeneous computers throughout this thesis.

Network distribution of P2G kernel instances requires a method of communication that enables us to transfer data between kernel instances running on different computers within a network. Network distribution also requires network communication between the P2G runtime on each computer and the scheduler. Our research includes these forms of network communication.

1.3 Limitations

Distribution of P2G kernel instances over all processing units on any type of heterogeneous computer is out of scope for this thesis. However, we can make design choices that simplifies adaptation of additional processing units. In this thesis, we focus on the following three types of processing units, with a primarily focus on the first two:

- GPUs programmable with Nvidia CUDA [71] or OpenCL [72].
- The x86 family of CPUs, including 64-bit CPUs.
- The Cell Broadband Engine (CBE).

We refer to these processing units as our *target architecture*. The term is used broadly, referring to computers with any combination of one or more of these

processing units as our *target architecture*. Furthermore, we refer to a network of computers with our *target architecture* as our *target global topology*.

Some processing units only support our *target architecture* on specific operating systems. The Intel OpenCL drivers [73] are only available for Microsoft Windows Vista and Windows 7. OpenCL drivers for the IBM BladeCenter [74] are only available on specific versions of Red Hat Enterprise Linux (RHEL). The SDK for the CBE is only available for certain versions of Fedora Linux and RHEL. We do not want to exclude several viable processing units, thus we try not to limit our research to a specific operating system (OS).

We do not investigate scheduling. However, it is a part of the execution pipeline of a P2G application. Therefore, we have considered the implications of a scheduler in broad terms in our work. Our work depicts a centralized scheduler, however the scheduler in a distributed P2G may not be a centralized entity. Integrating our work into P2G would require major changes to many parts of the P2G code base, including development of the runtime, and is therefore out of scope for this thesis.

1.4 Research contributions

During this master study, we have published an paper on the P2G framework [9], and submitted one paper on P2G to the *International Conference on Parallel Processing 2011*, and a demo paper on P2G to *ACM Multimedia 2011*.

Our network communication research led to the creation of a C++ socket communication library, named *Sevent*. This library supports communication of big data arrays that is required for communication of data between P2G kernel instances in a network. Sevent also supports complex data structures, such as C++ objects and vectors, that is practical for communication between the P2G runtime on each computer in the network and the scheduler. We have published this library under the BSD open source license. *Sevent* is already integrated into the recently initiated migration of P2G from a single computer framework to a framework supporting network distribution.

The result of our work with heterogeneous architectures is a C++ library, named

Olib, that can be used by the P2G project to load native binaries on our *target global topology*. The complexities of communication in *Olib* is handled using *Sevent*. *Olib* achieves our goal of finding a viable solution for network distribution of P2G on a network of heterogeneous computers in an efficient manner, and our research has revealed several possible ways of improving our solutions in the future. We have started working on a paper on *Olib* for IEEE ICPADS 2011.

How to get hold of the source code for our two libraries is described in Appendix A.

1.5 Outline

The rest of this thesis is organized as follows; Chapter 2 describes the P2G framework in greater detail. Chapter 3 explores solutions for program execution on heterogeneous architectures, and introduces *Olib*. In Chapter 4 we explore solutions for network communication in P2G, and describe our *Sevent* library. Both Chapter 3 and Chapter 4 begins with related work and alternative solutions before describing our solutions and why they are chosen over all the alternatives. In Chapter 5, we continue describing our *Olib* our research, more specifically, how we combine *Olib* with *Sevent* to provide the foundation for network distribution of P2G kernel instances. Finally, we demonstrate the viability of our research for multimedia workloads in Chapter 6 before we conclude our work in Chapter 7.

Chapter 2

P2G - Parallel Processing Graphs

The idea of P2G was born out of the observation that most distributed processing frameworks lack support for real-time multimedia workloads and that data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks such as MapReduce and Dryad.

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. This 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 [10]. Functional languages such as Erlang [11] and Haskell [12] and the event-based SDL [13], map well to task parallelism. Here 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 also support 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.

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 [14]. Languages that take advantage of the concept of single assignment include Erlang [11] and Haskell [12]. 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 [15] and SIFT [16] 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 [17, 18]. 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 [19] 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

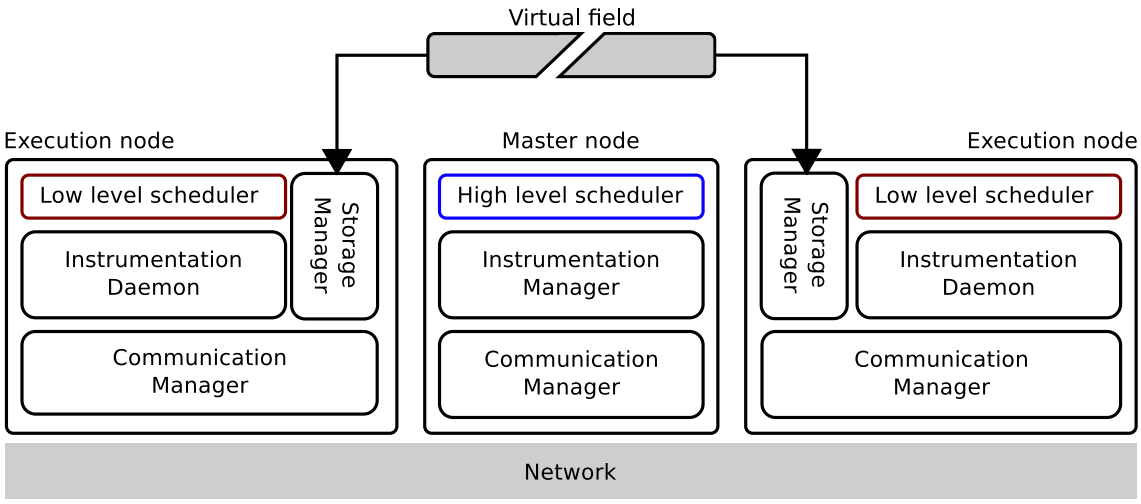

Figure 2.1: Overview of nodes in the P2G system.

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.

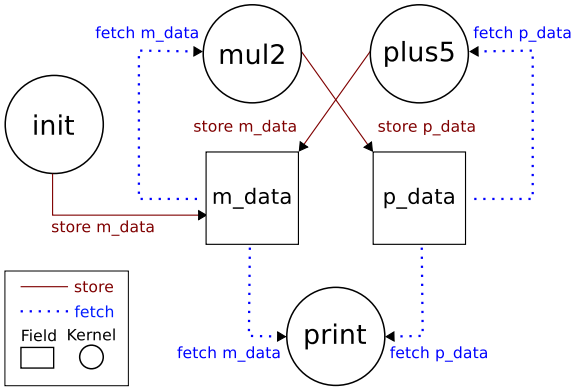

Figure 2.2: Intermediate implicit static dependency graph.

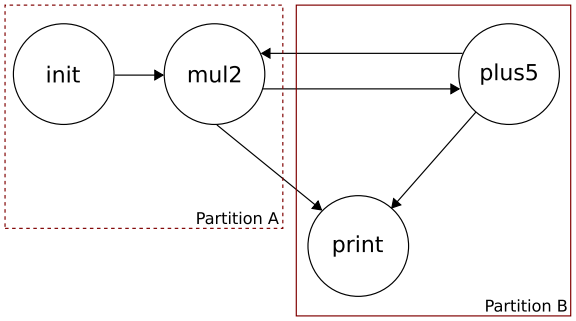

Figure 2.3: Final implicit static dependency graph.

2.1 Architecture

As shown in Figure 2.1, 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 store and fetch statements, from which it can generate an intermediate implicit static dependency graph (see Figure 2.2) where edges connecting two kernels through a

field can be merged, circumventing the need for a vertex representing the field (as seen in figure 2.3). From the intermediate graph, the HLS can then derive a final implicit static dependency graph (see Figure 2.3). The HLS can then use a graph partitioning [20] or search based [21] 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 2.3), an LLS at an execution node is responsible for maximizing local scheduling decisions. We discuss this further in section 2.2, but Figure 2.5 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. [22], 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.

Field definitions:

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

Kernel definitions:

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

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

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

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

Figure 2.4: Kernel and field definitions

Listing 2.1: C++ Equivalent of Figure 2.4

```
1  void print(int* data, int num) {
2      for(int i= 0; i < num;++i) {
3          std::cout < < data[i] < < " ";
4      }
5      std::cout < < std::endl;
6  }
7
8  int main() {
9      int data[] = { 10, 11, 12, 13, 14 };
10     int num = (sizeof(data)/sizeof(*data));
11     print(data, num);
12     while(true) {
13         for(int i = 0; i < num; ++i) {
14             data[i] *= 2;
15         }
16         print(data, num);
17         for(int i= 0; i < num; ++i) {
18             data[i] += 5;
19         }
20         print(data, num);
21     }
22     return 0;
23 }
```

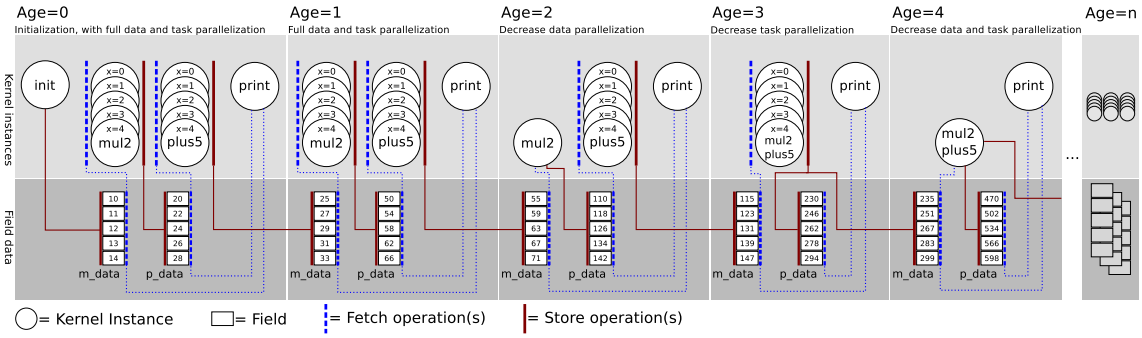

Figure 2.5: Dynamically created directed acyclic dependency graph (DC-DAG).

2.2 Programming model

The programming model of P2G consists of two central concepts, the *implicit static dependency graph* (Figures 2.2 and 2.3) and the *dynamically created directed acyclic dependency graph* (DC-DAG) (Figure 2.5). We have also developed a *kernel language* (see Figure 2.4), to make it easier to develop applications using the P2G programming model, though we consider this language to be interchangeable. Listing 2.1 shows C++ code that is equivalent to Figure 2.4. The C++ version must be executed sequentially, however the P2G kernel language version is segmented into separate kernels that P2G can execute in parallel.

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 2.5). 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.

2.2.1 Dependency graphs

The intermediate implicit static dependency graph (as seen in Figure 2.2) 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 2.3. 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 2.5. 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 2.5. 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 combin-

ing 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 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.

2.2.2 Kernel language

Considering the complexities of developing for 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, if not an impossible, task. We have therefore developed a *kernel language*. An implementation of our example workload is outlined in Figure 2.4.

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.

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 2.2.

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, as long as the age increases for each store operation (as seen in Figure 2.5). 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 2.1).

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 2.5). The scheduler makes its decisions based on the implicit static dependency graph and instrumentation data.

2.2.3 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, de-

pending on the number of fetch statements of the kernel definition. For example, looking at Figure 2.5 and 2.4, 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 2.4. 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 2.5 and 2.4 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.

2.3 Summary

P2G works on a single x86 CPU. The kernel language with implicit message passing and distribution makes programming for P2G a far less labour intensive task than comparable solutions. P2G does not support execution of kernel instances on all processing units in our *target architecture*. Furthermore, P2G does not sup-

port network-distributed execution. The rest of this thesis focus on solutions for these shortcomings, starting with heterogeneous architectures in Chapter 3.

Chapter 3

Supporting heterogeneous architectures using *Olib*

In this chapter, we explore alternative solutions to different parts of the challenge of creating a P2G application format usable on heterogeneous architectures. We specify our choices to each part of the challenge and compare each choice to alternative solutions, before explaining how we merge our choices into a working solution named *Olib*.

Distribution of P2G kernel instances over all processing units on our *target architecture* may be achieved in several different ways. We want P2G applications to automatically use the most efficient combination of available processing units. Some processing units can only execute a limited number of concurrent tasks. For example, NVIDIA CUDA devices with compute capability 1.0 can only execute a single task at any given time, and only some CUDA devices with compute capability 2.0 can execute multiple concurrent tasks [70]. Even on architectures such as multi-core CPUs, where we can start hundreds of parallel tasks, hardware properties, such as the number of CPU cores and the speed of shared memory, limits the amount of performance gained by increasing the number of concurrent tasks.

With this in mind, we can formalize our requirements for distribution of P2G kernel instances over all processing units in our *target architecture*. We require a method that allows P2G to use every available resource in our *target architecture* concurrently. Furthermore, this method must enable P2G to use the available

resources in a manner that achieves optimal performance for any given problem. We will clarify what we mean by *optimal performance* with an example. If we create an application that is able to be parallelized over all processing units in our *target global topology*, the kernel instances executed by this application must be able to use every available resource to such an extent that the performance of the application is limited by available processing power.

In this chapter, we discuss methods that can be used to meet our requirements. Furthermore, we present a method that we theorize can meet our requirements.

3.1 System virtual machines

System Virtual Machines (SVMs), such as XEN [23], KVM [24], VMWare [75] and QEMU [25], run isolated operating systems within another operating system, or on hardware designed to host SVMs. Running P2G on SVMs would enable us to run P2G on multiple operating systems on the same computer at the same time. This may enable P2G to use processing units that is supported on different operating systems concurrently on the same computer. There have been numerous performance studies on SVMs, including [26–31]. These studies show that operating systems running on SVMs in many cases achieve near-native performance.

SVMs may be usable by P2G, but they have sparse support for heterogeneous architectures. All hardware that is to be used by an SVM must be supported by the hardware abstraction layer in the virtual machine. There is sparse support for GPUs, and existing research [32] does not achieve close to native performance. In addition to sparse hardware support, P2G would have an additional delay while waiting for support for new hardware in SVMs, in addition to driver support in the underlying operating system before being able to use new hardware. Sparse support for heterogeneous architectures on SVMs limits their use to hosting the P2G runtime on multiple operating systems on computers with good hardware support in the SVM.

3.2 Process Virtual Machines

Process Virtual Machines (PVMs) run as a process inside an operating system, providing a platform-independent environment allowing a program to execute the same way on any platform. There are several PVMs, including the Common Language Runtime (CLR) [33] used by Microsofts .NET, and the Java Virtual Machine (JVM) [34]. PVMs can achieve performance close to native binaries for some workloads [35–38]. However, research on multimedia workloads is hard to find. A challenge with PVMs is programming language support. There are compilers for JVM and CLR for multiple programming languages [76, 77], but research on the performance of compilers for languages other than the most widely used languages is hard to find.

Another challenge is the limited support for many-core processors [39]. There has been research to overcome this challenge on CBE with CellVM [40,41], which allows JVM binaries to make use of SPEs. Unfortunately, CellVM is a research project without a stable code base. This makes it unsuited as a foundation for P2G. We do not have knowledge of any PVEs capable of using commodity GPUs in a platform independent manner. There are language bindings for OpenCL and CUDA for CLR [78–80] and JVM [42] [81]. Language bindings makes it possible to create platform independent applications on these architectures. But such applications are only portable to architectures with the supported GPUs. This means that we would need separate binaries for each GPU architecture, thus invalidating the advantage of allowing programs to execute the same way on any platform.

PVMs have positive properties, such as platform independence on homogeneous architectures, and performance close to native binaries on many workloads. However, they do not transparently make use of all processing units in our *target architecture*, and there is little research on the performance of PVMs on multimedia workloads. Close to native performance on many workloads is not good enough to satisfy our performance requirements. For PVMs to be a viable solution for P2G kernel instances, we would require research on multimedia workloads that proves that the performance of multimedia workloads on PVMs is indistinguishable from optimized native binaries.

3.3 Native binaries for any supported processing unit

A native binary is a program compiled into native machine code for a specific hardware architecture. It can be optimized for its target architecture, and thus achieve optimal performance. If we are to use native binaries for every processing unit in our *target architecture*, we require binaries for any combination of processing units in the *target architecture*. Solving this challenge is twofold.

First, we have to obtain implementations of our application with optimizations for any processing unit in our *target architecture*. The processing units in our *target architecture* do not provide the same programming interface. Thus, we would have to implement logically equivalent code in many different ways. This is time consuming and error prone. The P2G kernel language is designed to support multiple heterogeneous architectures. Development of P2G kernel language compiler support for our *target architecture* is out of the scope of this thesis, but we assume that it is possible.

The second challenge is binary loading. Since we want to achieve optimal performance, we need to load binaries for the optimal combination of processing units. Some processing units have strict limits to the number of concurrent tasks they can execute. Therefore, we have to use less optimal combinations of processing units if an application with higher priority requires the use of the most optimal processing units for our application. This also infers that subsequent executions of the same application do not have to use the same combination of processing units, and thus not the same native binaries. Further, it infers that for long running applications, more optimal performance might be achieved if we can switch to more optimal combinations of processing units at runtime.

We need to adapt dynamically to the available resources at runtime. We also need to be able to use more than one processing unit concurrently, and we require the ability to execute concurrent tasks on the same processing unit. Thus, a typical application should do the following at runtime:

- Start native binaries that make use of the optimal combination of available processing units.
- Send data between the binaries.

- Start and/or stop binaries if more optimal combinations of processing units become available.

Using native binaries on a heterogeneous architecture is complex. However, by using native binaries, we can, in theory, achieve optimal performance with a perfect scheduler. SVMs and PVMs can not provide the performance and processing unit support required by P2G. Therefore, we will focus our research on using native binaries to execute kernel instances in P2G.

3.4 An application format for heterogeneous architectures

If a P2G application is to consist of native binaries for each processing unit in our *target architecture*, we need a way of organizing these binaries into an application.

Fat binaries [82] bundle binaries for multiple architectures into a single binary. Users can obtain a single binary that will run on any hardware architecture compatible with one of the bundled binaries. This method is called Universal Binaries in recent versions of Apple Mac OS X [83]. Universal Binaries use the same underlying binary format, Mach-O [84], as used in NEXTSTEP for fat binaries. There has been an attempt at incorporating fat binaries into the GNU Linux toolchain with the FatELF project [85], but it has not been widely accepted, and the development has stalled [86]. According to the FatELF project website, fat binaries has minimal runtime overhead compared to normal native binaries. However, they do not provide any peer reviewed research to back up this claim.

Fat binaries could serve as an archive format for all the binaries required by a P2G application. However, operating system support is lacking. Additionally, we believe that a single archive containing binaries for any processing unit is suboptimal for P2G. Our *target architecture* contains many processing units that are rarely used on the same computer. For example, one would rarely have an AMD CPU and an Intel CPU on the same computer. A more optimal solution should enable us to distribute a minimal number of binaries to each computer in our network, to reduce network load and storage space requirements.

The Java Runtime Environment (JRE) supports an application file format based on the ZIP file format, named JAR [87]. The JAR format stores all files related to a Java application in a ZIP-file, with additional files containing meta-information, such as how to start the application. When the JAR-file is executed through the JRE, the Java application within the JAR-file is executed, and the JRE provides transparent access to the resources within the JAR-file.

Another similar technique, used in Apple Mac OS X, is loadable bundles. Loadable bundles are packages of executable code and related resources that can be loaded at runtime [88]. Mac OS X uses a directory structure where they put binaries and related resources. The directory can be moved anywhere, and when the directory is executed, an executable defined in a configuration file in the bundle is executed.

We propose to adopt the Loadable bundle approach for P2G. We can put all binaries for a P2G application in a directory. Furthermore, we only need to install a subset of these binaries on each computer in our network, and updates to one binary only require us to re-install that binary. Such a format can become hard to provide as a download from static sources such as a website. Therefore, we recommend future study into a format similar to JAR to complement a directory-based format.

3.5 Binding native binaries to processing units

Having a directory of binaries is little help without any way of knowing what binary is most suitable for a processing unit. An instrumentation component of P2G might gather information about each computer in a network distributed version of P2G. We have no control over the level of detail such an instrumentation component may provide. Since we have no control over the level of detail, we consider this out of the scope of our thesis. However we have considered how to integrate this information into our application format. With a directory based application format, we can add a configuration file for each native binary. These configuration files can describe usable processing units for the corresponding native binary, using a level of detail that the instrumentation component can provide.

3.6 Native binary loading

So far, we have discussed binaries optimized for specific hardware architectures as native binaries. A native binary is a file format containing machine code for a specific architecture, and any other information required by the binary loader to execute the machine code. A native binary does not have to be an executable application, it can also be a part of an executable application, or machine code that can be used by many applications concurrently. There are many file formats for native binaries, such as the Executable and Linkable Format [43, 44] and Portable Executable [45]. We have not found any research into performance gains of selecting a specific native binary file format, and have therefore opted to use the de-facto native binary format on each operating system.

We have established that we will research P2G applications as directories of multiple native binaries. We assume that the P2G scheduler will load the appropriate binaries when we execute an application using the P2G runtime. File formats for native binaries have two ways of loading native binaries at runtime with wide operating system support.

3.6.1 Separate processes

The P2G runtime can execute each native binary as separate processes. These processes have to communicate with each other, and with the P2G runtime. There are several methods for inter-process communication (IPC).

Any form of IPC must consider the binary representation of data types. Compilers do not handle binary representation of data in the same manner. Some programming languages do not specify exactly how compilers should represent data types as raw bytes. For example, the C programming language, specifies that the native type *int* has to be at least 16 bits long [46]. The compiler is free to represent ints using more than 16 bits. When two processes communicate, they have to use the same binary representation of data types, or provide an abstraction layer which takes care of translation between different representations.

A software pipeline is a simple form of IPC. A process has standard input and output streams. On most operating systems, processes can communicate by writ-

ing to the standard input stream of another process, which in turn, can read from its own standard input stream. Software pipelines are inflexible since each process only have one input and one output stream.

A more flexible method of IPC is shared memory. Multiple processes can access the same pages of memory. An issue with this scheme is *cache coherence*. In multi-core systems, it is common for each processor to have its own memory cache. This cache allows processes to access frequently used pages of memory fast. However, when a process writes to a cached memory page, data is written to the cache instead of to RAM. If we have multiple caches, and two communicating processes use different caches, they would both manipulate data in their respective cache, which is unavailable to the other process. POSIX systems provide the shm_open function which handles cache coherence automatically. An issue with shared memory is security. On POSIX systems, any process can access shared memory initialized by another process.

A more complex but very portable method of IPC is message passing, where processes communicate messages containing data, such as raw bytes or complex data structures. There are many message passing frameworks, including CORBA [47], MPI [48], D-BUS [49] and SOAP [50]. Most message passing frameworks automatically handle differences in binary representation of data types, often at the cost of limiting what data types they can communicate.

3.6.2 Dynamic loading of shared libraries

Shared libraries are native binaries that can be loaded at run-time and used by many applications concurrently. They are loaded into a physical page in RAM, and any application using the library executes this page, and maps the page into its own address space. A shared library can be loaded from a storage device into RAM once, and used concurrently and multiple times by processes. From a programmers perspective, a shared library behaves like a statically linked library after it has been loaded. However, the binary loader in the operating system provides this abstraction, and programmers have no control over how this abstraction is provided.

The P2G runtime can load P2G kernel instances as shared libraries into its address

space at runtime, and call functions within the libraries to execute the kernel instance. The P2G runtime needs to load multiple kernel instances, and use them concurrently. Threads can solve this challenge. With each kernel instance running in its own thread within the P2G runtime, kernel instances can communicate directly within the address space of the P2G runtime.

If the P2G runtime is to be able to load kernel instances as shared libraries, the shared libraries have to be compatible with the P2G runtime. Most low level programming languages, such as C, C++ and Fortran can load shared libraries compiled from other programming languages. However, there are limits to this interoperability. Programming languages have different ways of representing data types, and they do not all have the same set of supported data types. Additionally, as explained in the previous section, the binary representation of data types differ between compilers.

3.6.3 Conclusion

We do not believe binary incompatibility of processes or shared libraries used on the same computer on our *target global topology* is an issue, because we assume the P2G runtime and kernels would use the same compiler. Both processes and shared libraries are viable solutions for P2G kernels. We believe each approach merit further study. Section 3.7 contains details about an implementation of the shared library approach, thus the separate process approach is possible future work.

3.7 Obese libraries on a single computer

We have merged the chosen solutions discussed in the previous sections in this chapter into a C++ library, named *Obese libraries* (*Olib*), which uses directories of shared libraries to provide P2G applications. The name is a reference to its similarities with fat binaries, and we find the obese part of the name fitting since a heterogeneous architecture will encourage binaries containing code for more architectures than a homogeneous architecture.

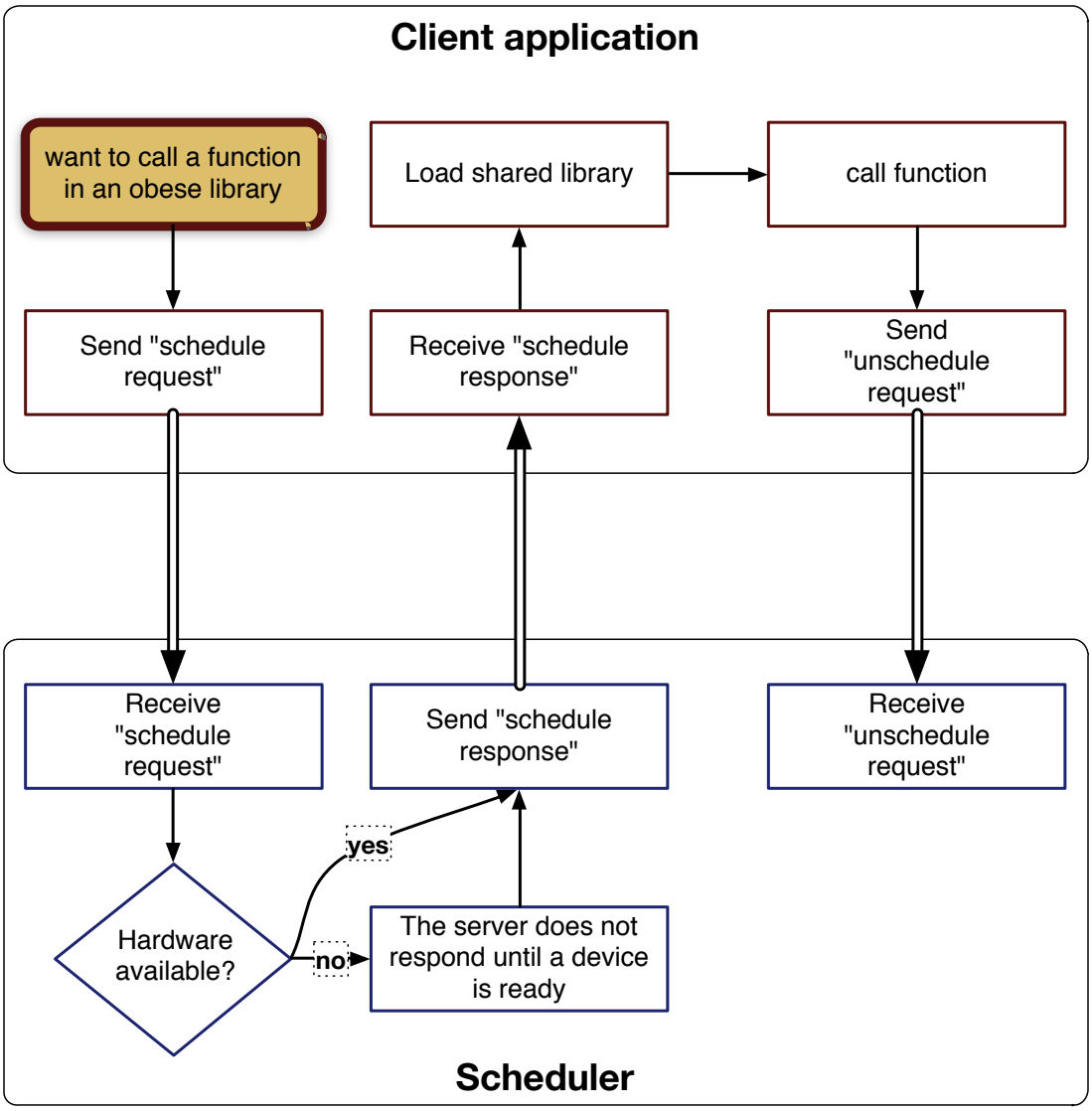

Figure 3.1: Flow of requests and responses in the single computer *Olib* implementation.

Scheduling and instrumentation is, as explained previously, out of the scope of this thesis, therefore our implementation uses a very simple static scheduler. To assert the viability of our solution without unneeded complexity, we choose to make the framework usable on a single computer before considering network distribution.

Figure 3.1 shows how requests and responses flow between request and response handlers in the single machine *Olib* implementation, and Handler-listing 1 de-

Schedule request handler receives *scheduling requests* containing the file-system path to an application directory and the name of a function which every shared library within the application directory must provide. The scheduler responds with the file-system path, the name of a shared library located within the application directory, and the name of the function. The name of the shared library is selected by the scheduler from a list of shared library names in a round-robin manner. We assume that a scheduler integrated into P2G would use a more dynamic method of choosing this name using instrumentation data.

Schedule response handler receives responses to scheduling requests, loads the given shared library, and invokes the given function within the shared library in a thread. When the execution of the function completes, the scheduler is notified using an *unscheduling request*.

Unscheduling request handler receives *unscheduling requests* containing the file-system path to an application directory, and the name of a shared library within the application directory. This is enough information for the scheduler to know that the hardware is used by one less kernel instance.

Handler-listing 1: Handlers in single computer *Olib*.

scribes each handler in detail. A client application sends a scheduling request, receives a response, and loads the shared library that is part of the response from the scheduler. The scheduler is deployed as a server, typically with the same uptime as the operating system where it is deployed, while a client application typically performs a specific task, and quits. As stated previously, we do not implement the P2G runtime, however the functionality required by the P2G runtime to integrate *Olib* is demonstrated through the client application.

3.7.1 Loading shared libraries at runtime

We load shared libraries using the *dyld* library on Mac OS X and GNU Linux. Our implementation only supports these platforms, but other operating systems have similar methods for shared library loading at runtime, such as the `LoadLibrary`-function on Microsoft Windows [89].

Listing 3.1: Source code of the utils library

```
1 #include "utils.h"
2 int sum(int a, int b) {
3     return a + b;
4 }
```

Listing 3.2: Load a shared library at runtime

```
1 #include <dlfcn.h>
2 int main(int argc, const char *argv[]) {
3     typedef int(*sum_t)(int, int);
4     void* lib = dlopen(SHAREDLIB_PATH, RTLD_LAZY);
5     void* sumPtr = dlsym(lib, "sum");
6     sum_t sum = (sum_t) sumPtr;
7     int result += sum(1, 2);
8     dlclose(lib);
9     return 0;
10 }
```

Listing 3.2 shows how to load the shared library in Listing 3.1 at runtime. On line 4, we load the shared library from the filesystem path, SHAREDLIB_PATH. The next line locates the memory address of the `sum`-symbol in the library. On line 6, we cast the `symbol`-pointer to our function type. On line 7 we use the `sum`-function. This works because the shared library has been loaded into the address space of the application, and therefore the `sum` function can be executed from the address returned by the call to `dlsym` on line 5. On line 8, we close the shared library, signaling that the underlying system can unload the library from RAM. Exactly what happens when we call dlclose is implementation specific. For example, on Mac OS X 10.5, the library may be kept in cache even if no application is using it [90].

The examples above are in C. Since P2G uses C++, we need to be able to load shared libraries written in C++. Functions in C++ can not, by default, be referred to uniquely by their name. C++ functions support a varying number of parameters through multiple implementations of functions with the same name, but with different parameter types [51]. Therefore, function names in C++ do not refer to a specific function. To provide functions with unique names, their names are mangled [52] into unique names by the compiler. There is no standardised name mangling scheme for C++, and the Annotated C++ Reference Manual [53]

encourages the use of different name mangling schemes.

C++ is designed to interoperate with C programs and libraries that require a unique name for a function within any scope [46]. Listing 3.3 demonstrates how we can define our functions as extern C functions to give them a unique name. As demonstrated through the use of the `Person` class, declaring a function as extern C does not restrict the us of C++ specific language features. However, we can not provide multiple functions with the same name within an extern C block.

Listing 3.3: Making a C++ function loadable by dlsym

```
1  #include <string>
2
3  class Person {
4      public:
5          std::string name;
6          unsigned age;
7          Person(const std::string& name_, unsigned age_) :
8              name(name_), age(age_) {}
9  };
10
11 extern "C" {
12     Person createPerson(const std::string& name,
13                         unsigned age) {
14         return Person(name, age);
15     }
16 }
```

3.7.2 Live patching

It may not always be possible to halt, patch, and resume a running application because of deadlines. Patching an application is an operation that changes certain parts of an existing application. Live patching does such changes at runtime. Since the scheduler determines what shared library to load, we can use shared libraries for live patching of long running applications. In a version of P2G with a dynamic scheduler, we can distribute a shared library that fixes a bug, improves performance, or adds support for additional processing units, and instructs the scheduler to use the new shared library instead of, or in addition to, the old one.

The application does not have to halt, it will only have to be told of the new shared library.

3.7.3 Communication with the scheduler

Since scheduling is out of scope for this thesis, we do not optimize communication with our scheduler. We planned to make *Olib* distributed over the network, therefore, we chose to use socket communication with the scheduler. *Olib* uses a plain text protocol on top of the TCP protocol for all communication. Our requests are encoded as plain text before sending, and decoded upon arrival. Even with our simple communications, a plain text protocol on top of TCP is non-trivial to change, thus impeding our ability to research all our alternatives effectively.

3.8 Summary

We have demonstrated that loading and using directories of shared libraries for P2G kernel instance is a viable solution. Furthermore we have highlighted several possible alternatives and improvements to our current solutions, such as separate processes instead of shared libraries, support for additional operating systems, and an addition to our application format to enable the ability to download an application as a single file.

For a distributed version of *Olib*, we require a method for communication between P2G kernel instances running on different computers. A distributed version of *Olib* requires changes to the various components of *Olib*, and how they communicate. Our experience with scheduler communications, demonstrates that we could benefit from another method of expressing communication. Since communication is such an important part of distributing *Olib*, we have dedicated the next chapter to this topic. We continue discussing our *Olib* research in Chapter 5.

Chapter 4

P2G network communication using *Sevent*

We introduced the complexities of socket communication, and why we need it, in Chapter 3. Integrating a distributed version of *Olib* into the P2G framework requires changes on many levels in the framework. Most of these changes have to do with the fact that P2G must be able to communicate in a network.

We could ignore any communication needs except the requirements of a distributed *Olib*-implementation. This may enable us to make a less complex communication library. However, any other component of P2G that requires network communication would also have to solve the challenges of network communication. This may lead to multiple socket communication libraries with overlapping functionality, or patches to our library. Overlapping functionality would make P2G harder to maintain. Patches to a library specifically written for communication between Obese libraries may work, but it is unlikely that all design choices made with only Obese libraries in mind would fit the overall needs of P2G.

In this chapter, we will deduce the communication requirements for P2G, and describe a modular and flexible socket communication library, named *Sevent*, meeting these requirements.

4.1 Requirements

P2G kernel instances can run on any computer in our *target global topology*, and they must be able to communicate. Kernel instances should be able to communicate data directly to communicate data between store and fetch statements. Other components of the system that requires communication, is scheduling and instrumentation. Fetch and store data is typically big arrays of a native data type, and meta information such as the size of the array and where the output should be sent. Instrumentation data is detailed information about hardware in the network, needed for the scheduler to decide which computer in the network, and which specific PU on a computer, is most suitable for any given application.

P2G needs to be able to send data of common types. It is hard to specify our exact requirements without full knowledge of every workload, but it is safe to assume that the following data should be supported:

- Signed and unsigned integers of various size. Almost anything can be represented as integers of various size.
- Floating point numbers of various precision. Complex calculations often require floating point arithmetics, and it would be suboptimal to exclude data communication in to and out from such calculations.
- Complex data structures such as vectors, maps and class hierarchies.

Complex data structures are not absolutely required to communicate in P2G, but they make it easier to write clean, readable and maintainable code. For example, consider communication of instrumentation data. Instrumentation data could be packed as a string. A format where pairs of key and value are separated by semicolon, and each pair is suffixed by newline would work. The code to pack and unpack the string would be not be very complicated, but we would need special handling of newline and semicolon.

Some parts of the instrumentation data is stored in complex date structures. There are lists of storage devices, and each entry in the list is a map of properties. Some of these properties are numbers. Consider the added complexity of supporting values which themselves are complex data structures, or of various data types. The separators on each level will have to be chosen carefully, and we would need

a way of knowing the data type of each value. One may end up writing custom serialization for each message type. Serialization is just one of many examples where complex data structures are required. If we limit the data types supported for this communication, we effectively limit the kind of tasks we can distribute with obese libraries. Custom solutions for serialization of complex data structures would increase the overall complexity of P2G, since we would have a multitude of solutions to maintain instead of one. This added complexity would make P2G harder to stabilize and maintain. We conclude that we need to support serialization of complex data types.

4.2 Serialization

There are several good serialization libraries available for C++. Boost serialization [91] supports serialization of any C++ type, even complex types such as vectors and maps. Google Protocol Buffers [92] take a different approach. They let the programmer define a data format using their own language. This data format definition can then be compiled into source code for many different programming languages, including C++. Apache Trift [93] works in the same way as protcol buffers. External Data Representation (XDR) [94] is an Internet Engineering Task Force (IETF) standard from 1995, which allows a specific set of data types to be wrapped in an architecture independent manner. Since XDR, Protocol Buffers and Trift only support a very specific set of data types, they can optimize their binary format for speed and compression. Boost serialization is much more flexible, and meet all of our requirements for a serialization library. The speed and compression of Boost serialization depend on the archive format. Boost serialization supports compressed and fast binary archives, and uncompressed and slow plain text archives. Boosts official binary serialization format is not endian safe, however there is an endian safe binary archive in their source code repository.

P2G needs the flexibility of Boost serialization, and we also need efficient transport of arrays of native data types. Our conclusion is that no serialization library meets all our requirement. Thus, we need to make it possible to use the library best suited in each case. This may lead to complex code, so we need to make an API which minimizes this complexity.

4.3 Communication protocol

All communication in the system needs to be reliable and efficient. For reliability, Transmission Control Protocol (TCP) is a natural choice. An approach using User Datagram Protocol (UDP) may lead to better throughput with small messages, since the overhead of establishing a connection, startup and sequential transfer is avoided. But, it would require a much more complex application layer communication protocol, since we would need to make reliability a part of our application layer protocol. Some network hardware guarantee reliability, and would therefore be able to achieve higher throughput with a custom UDP based application layer protocol. However, such optimizations are outside the scope of this thesis. For big messages the overhead of TCP would be far less noticeable. There is much recent research on this topic. Several new protocols have been suggested, and there have been numerous improvements to the TCP protocol [54–56]. Until we can prove that the choice of a more complex solution would improve the overall efficiency of the entire framework, we choose to use TCP.

TCP communication is usually achieved using socket API, such as Berkeley sockets. Berkeley is far too complex to work with directly, when we only need a small subset of its features. Thus we would benefit from a higher level method of communication.

4.3.1 Message passing interface

The Message Passing Interface (MPI) [48] is a language independent communication protocol coupled with a process distribution framework. There are multiple versions MPI, and many different vendors provide their own implementations. MPI provides a virtual topology, synchronization, and communication between processes. P2G provides the same features, but MPI is designed to distribute processes in a homogeneous topology, while P2G is designed to distribute finer grained parts of an application in a heterogeneous topology. While MPI is a stable and widely used API, it is a process oriented approach for a homogeneous environment, and thus not compatible with our requirements.

4.3.2 CORBA

The Common Object Request Broker Architecture (CORBA) [47] is a standard that enables programs written in different programming languages to communicate. CORBA supports communication between programs on the same machine, and between programs on multiple computers. CORBA uses an interface definition language (IDL) to described the objects used in communications. Languages such as Java and Python are easy to map to IDLs. However, the C++ language mapping for CORBA is difficult to use because the API is complex, and error prone, especially with respect to thread safety, exception safety and memory manangent [57]. These issues makes CORBA unsuited for P2G.

4.3.3 D-Bus

D-Bus [49] is an inter-process communication system designed for communication between applications in desktop systems. Applications can register objects with the D-Bus daemon, and other applications can communicate with these objects. D-Bus has many attractive properties for P2G, such as asynchronous communication and cross-platform communication, however P2G requires support for complex data types. D-Bus can send multi-part messages, however these messages are limited to specific set of data types.

4.3.4 Remote Procedure Call

Remote Procedure Call (RPC) is a method of communication enabling procedures within another address space to be called. The term *procedure* is used broadly in this context, meaning function, method, or subroutine. This address space may be within another process on the same computer, or within a process on another computer on the same network, such as any computer connected on the Internet.

Our focus on high performance leads us to conclude that the RPC approach is too inefficient. A procedure call is a blocking operation, thus making the calling thread wait for the remote call to finish and transfer of response data to complete.

This issue could be solved using a thread for each remote procedure call. However, in a system with many small messages this will lead to a huge number of threads. A thread pool would be able to limit the number of threads, but then we would run the risk of deadlock when a running thread waits for data from a thread which has not yet been scheduled. All these issues could be solved using asynchronous communication.

It is possible to implement an asynchronous RPC, and the RPC term is often used more generally to describe any method of invoking processes in another address space. This confusion in the way the RPC is defined, makes us conclude that we do not wish to the express the communication within P2G using the RPC terminology. Using a terminology with multiple definitions would make it hard to understand and discuss communication within P2G, and misunderstandings would be very likely to occur.

4.4 Initial design: *P2G-RPC*

Since no solution fulfilling all our requirements of a socket communication library exists, we need to create our own. At an early stage of the development of P2G, prototypes of instrumentation and simple distribution was in progress, and their exact requirements were still under development. At this early stage, our first communication library was developed. Unspecified requirements led us to design a complicated library, called *P2G-RPC*, that did not fit our needs perfectly.

The design of *P2G-RPC* is outlined in Figure 4.1. Socket multiplexing is handled by worker threads polling data from the queue of incoming and outgoing data within the `IOService`. Worker threads look up the correct handler, and invoke the handle-method in `ServerHandler` or `CommunicationHandler`. On a listening server, `ServerHandler` is responsible for new connections from clients. The establish_connection-method in `Connector` establishes a new connection from a client. Each `CommunicationHandler` handles a single connection, and it does not know if the `ServerHandler` or the `Connector` initiated the connection.

The `SenderPool` is a queue of outgoing messages. A message contains the ad-

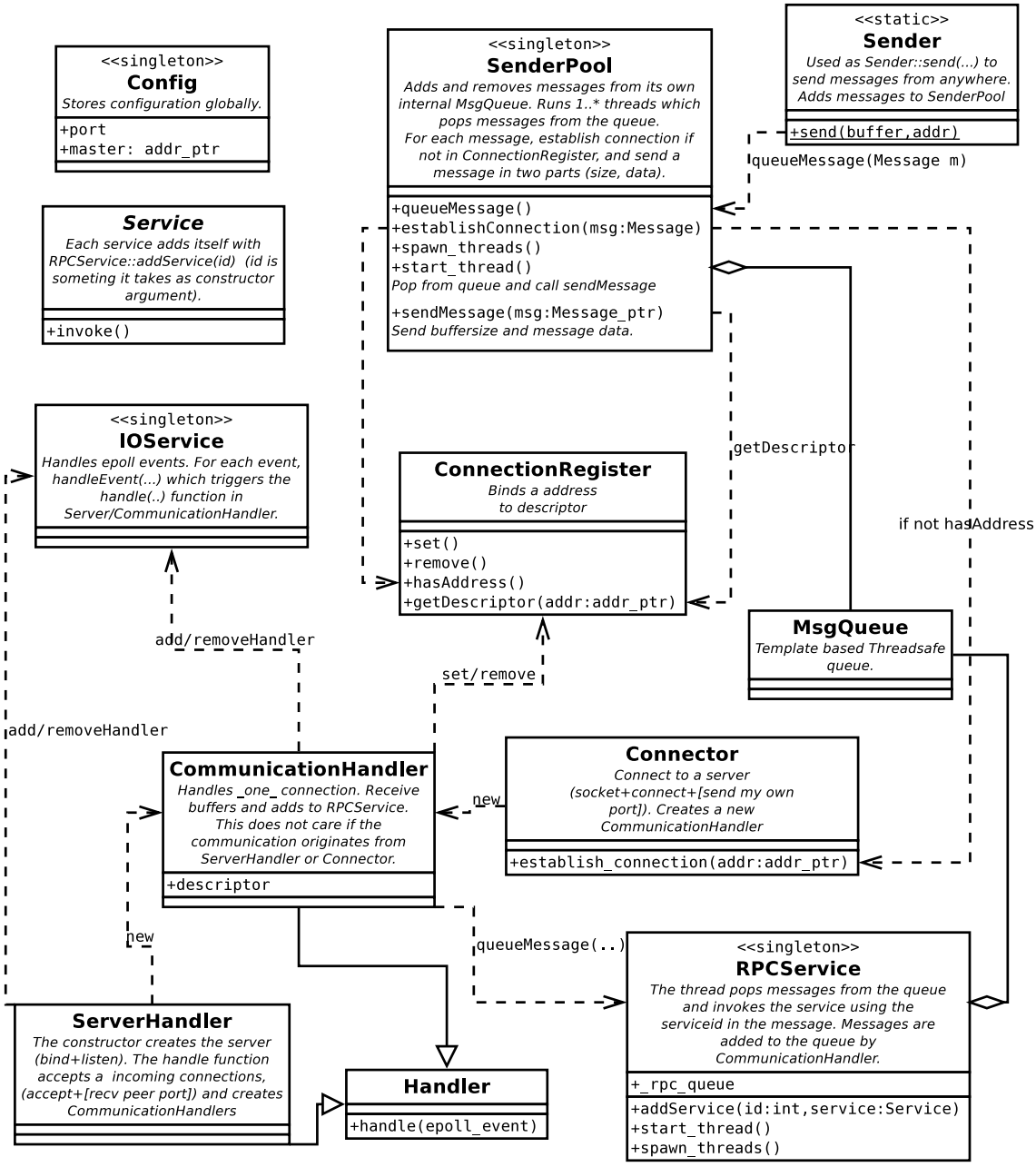

Figure 4.1: UML diagram of the *P2G-RPC* of the socket communication library.

dress of the receiver, and the `SenderPool` establishes a new connection to the receiver, unless the receiver is in the `ConnectionRegister`, which is a cache of open connections.

When `CommunicationHandler` receives a message, it adds the message to a queue in `RPCService`. Several threads pop messages from this queue and invoke user defined services. These user defined services are subclasses of `Service`.

Clients and servers are uniquely identified through an address, which consists of an IP-address and a port. Each message sent in the framework looks up this address in the `ConnectionRegister`, and creates a new connection if needed. This is a design choice that greatly simplifies communication since the communicators do not have to know if they are a client or a server, or even if they are connected. The cost of looking up the address in a map for each communication should be unnoticeable compared to the cost network communication.

Listing 4.1: Socket client example - *P2G-RPC*

```
1  struct HelloResponseHandlerService: public Service {
2      HelloResponseHandlerService(int id) {
3          RPCService::getInstance()->addService(id, this);
4      }
5      virtual void invoke(Message_ptr msg) {
6          std::string response;
7          msg->msg >> response; // Deserialize
8          std::cout << "Response: " << response << std::endl;
9      }
10 };
11
12 int main(int argc, const char *argv[]) {
13     const char* masterip = "127.0.0.1";
14     const char* masterport = "2000";
15     int port = 3000;
16     IOService* service = IOService::getInstance();
17     new ServerHandler(port);
18     SenderPool::getInstance()->setMessageSenderCallback(
           msgSenderCallback);
19     Config::getInstance()->setPort(port);
20     Config::getInstance()->setMaster(
21         addr_ptr(new addr_type(masterip, masterport)) );
22     RPCService::getInstance()->spawn_threads(1);
23     SenderPool::getInstance()->spawn_threads(1);
24     new HelloResponseHandlerService(
           SIMPLE_RESPONSEHANDLER_SERVICE_ID);
25
26     std::string hello("hello");
27     Buffer_ptr buffer( new Buffer(BUFFER_ENCODE,
28                                   "isi", hello.size()));
29     // Serialize and send
30     buffer << SIMPLE_RECEIVER_SERVICE_ID << hello << 42;
```

```
31      Sender::send(buffer, Config::getInstance()->getMaster());
32
33      while(true) { // Event loop
34          int events = service->waitForEvents(
35                      IOService::TIMEOUT_INSTANT);
36          for(int eventnum=0; eventnum<events; eventnum++) {
37              service->handleEvent(eventnum);
38          }
39      }
40  }
```

Listing 4.2: Socket server example - *P2G-RPC*

```
1  struct HelloReceiverService: public Service {
2      HelloReceiverService(int id) {
3          RPCService::getInstance()->addService(id, this);
4      }
5      virtual void invoke(Message_ptr msg) {
6          std::string question;
7          int i;
8          msg->msg >> question >> i; // Deserialize
9          std::cout << "Received: " << question
10             << ":" << i << std::endl;
11         std::string response("Hello world");
12         Buffer_ptr buffer(new Buffer(
13             BUFFER_ENCODE, "is", response.size()));
14         buffer << SIMPLE_RESPONSEHANDLER_SERVICE_ID
15             << response;
16         Sender::send(buffer, msg->addr);
17     }
18 };
19
20 int main(int argc, const char *argv[]) {
21     int port = 2000;
22     new ServerHandler(port);
23     IOService* service = IOService::getInstance();
24     Config::getInstance()->setPort(port);
25     RPCService::getInstance()->spawn_threads(1);
26     SenderPool::getInstance()->spawn_threads(1);
27     new HelloReceiverService(SIMPLE_RECEIVER_SERVICE_ID);
```

```
28      while(true) { // Event loop
29          int events = service->waitForEvents(
30                          IOService::TIMEOUT_INSTANT);
31          for(int eventnum=0; eventnum<events; eventnum++) {
32              service->handleEvent(eventnum);
33          }
34      }
35  }
```

Listing 4.1 and Listing 4.2 show how the library works from the programmers point of view. On lines 26-31 in Listing 4.1, the client creates and sends a message containing a string and a integer to the server. In the invoke-method on lines 5-17 in Listing 4.2, the server receives the message and responds with a string. The client in Listing 4.1 receives the response in `HelloResponseHandlerService`. These examples show that simple one-way communication requires a high amount of code.

4.5 Issues with *P2G-RPC*

Use of *P2G-RPC* in our *Olib* research, and in the instrumentation component of P2G revealed several issues with the design of *P2G-RPC*.

4.5.1 Unneeded complexity

The example in Listing 4.1 and Listing 4.2 shows message passing between two endpoints, which raises a question about unneeded complexity in the external API. This is a simple operation, and the API should reflect this simplicity:

Server

- Starts listening on a specific IP-address and port.
- Registers a callback listening for the request.
- Waits for all threads polling socket events to finish their work.

Client

- Connects to the server.
- Registers a callback listening for the response.
- Waits for all threads polling socket events to finish their work.

In a fundamental library that is designed to be re-used, the simplicity of the external API is important. The risk of bugs in the code using a library increases with the complexity of using the library. A simple external API does not have to limit the internal complexity or flexibility of the library. The library only needs to provide a higher level interface tailored for the most common usage pattern. This kind of higher-level interface is called a Facade [58]. Those few scenarios where such a Facade is too inflexible can be handled through a lower level API, such as the one in Listing 4.1 and Listing 4.2. If a scenario not covered by the Facade should become more common than first assumed, another Facade can be created.

4.5.2 Error handling

We can also deduce another important shortcoming from these examples. There is no apparent way of handling errors. Debugging of multi threaded network applications is complicated, and bad error handling makes it even more complicated. We spawn threads, but do not have any control over their errors. Since these threads handle socket communication, errors such as links going down, issues when establishing a connection, and messages too big to fit in memory, must either crash the program or be silently ignored. Logging of these errors does not really improve these matters, since many such errors have to result in some action that ensures that the error does not happen again. Another issue with silent errors is testing. One can not write automatic tests for silent errors.

4.5.3 Global state with Singletons

Automatic testing, such as unit testing, revealed another problem with the design, i.e. the use of global state through the Singleton pattern [58]. A Singleton is a design pattern that ensures that a class only has one instance, and provide

a global point of access to it. This essentially makes a Singleton a process global variable. The motivation behind the Singleton pattern is to ensure that a class has only one instance in situations where this is required. For example, some GPUs can not be used concurrently. If all communication with such GPUs are performed through a Singleton that disallows concurrent access to the GPU, we can assure that the GPU will not be used concurrently. *P2G-RPC* used Singletons in places where such restrictions are not required. Misuse of Singletons led to several issues.

Services register themselves with the `RPCService`-Singleton in Figure 4.1. From an external perspective, it is impossible to know what parts of the library actually use these services, since any part of the library may access them through the globally accessible `RPCService->getInstance()`. Maintaining tests involving global state is error prone, since any new addition to the library may change the result of any test. One of the reasons for writing tests is safety when extending and changing code. If changes to some part of the code change the result of unrelated tests, one can not trust these tests. If the `RPCService` had been a normal class, objects of this class within a scope defined by the user, the user would know what classes or functions depended on these services since they would need a `RPCService`-object as argument. This not only makes it possible to write isolated tests, but it also enables the user of the library to trace the workflow of the library to a much greater extent. Tracing the logical workflow of a library is very important when dealing with threaded applications, since misunderstandings may lead to deadlocks and starvation.

The general issue of how singletons are used is *tight coupling*. This is apparent with the `Config`-singleton in Figure 4.1. The `Config`-class makes configuration globally available, thus tightly coupling many classes within the library with the `Config`-class. Not only does this make it hard to trace the workflow of applications using the library, tight coupling also makes it very hard to write good tests. Many tests need to initialize mock-up data [59]. With tight coupling, we need to make mock-ups for every component coupled with the component we are testing, or we can make less fine grained tests where we test every coupled component as one. Each approach results in complex tests that are hard to verify.

All these issues related to testing and tight coupling may be ignored if the library works and never has to be changed. It is possible to write complex tests, work

out any bugs, and never touch the code again. However, when developing a library where requirements may change as a result of further research, this kind of approach is unrealistic.

Through further research with our framework for distribution of computationally intensive tasks, we realized that we need to be able to use multiple network connections between the same computers in a network to maximize throughput. Tight coupling of the `Config`-class with several other components made this change a major rewrite. The global `Config`-object contains the listening port. Every message to a single process will have to be received on this port. We should have been able to create listeners on two separate ports within the same process, but in *P2G-RPC*, we can only configure a single global port. We would also prefer to cleanly separate handlers on each of these interfaces. However, the `RPCService` where we register handlers is a Singleton, making this impossible.

4.5.4 Graceful shutdown

P2G-RPC has no way of executing any kind of shutdown. The worker threads run until the program is killed. With no way of shutting down our threads, automatic tests become very hard. We believe it is important to stress that this does not only make it very hard to write test for our framework, but it also makes it hard to get full test coverage in any application using the library.

When debugging, it is often useful for the threads to crash at the first error, while a production environment may handle certain errors, or at least shut down gracefully. A graceful shutdown should let any active handlers complete their work, and refuse subsequent incoming data and connections. Closing a socket is an error prone action. One peer closes the socket, and any thread within any of the two peers reading or writing the socket will finish with an error. With graceful shutdown, only one of the peers fail with an error, and it will only fail at the beginning of a read, write or connection. In conclusion, graceful shutdown simplifies debugging through better error messages and support for automatic tests.

4.5.5 Berkeley socket API and Epoll

The Berkeley socket API is a low level API for socket programming. Implementations vary a bit between platforms, making it a bit hard to write portable Berkeley socket code. There are many methods for efficient socket multiplexing. A very efficient, Linux only implementation, is Epoll [95]. We chose Epoll for *P2G-RPC* for its efficiency. Further research revealed several libraries that makes Epoll available on Linux, and other implementations with comparable performance available on other platforms through the same API. Two widely used such libraries are libevent [96] and Boost ASIO [97]. The P2G project already depends on the Boost library. Boost ASIO has a good API that is low-level enough for us to optimize communications if required, and it has higher level interfaces for common operations. Libevent has many of the same qualities, however it is written in C. In a C++ codebase, a good object oriented library, such as Boost ASIO, integrates far better. Since nothing else significantly distinguishes these libraries, Boost ASIO is a natural choice.

4.5.6 Thread API

POSIX threads is a powerful and portable threading API written in C. The latest working draft of the next standard for the C++ language, dubbed C++0x, contains a new thread API. This API integrates much better into a C++ codebase such as P2G. The new thread API in C++0x [98] is not yet widely implemented by compilers, but the Boost thread [99] implements the API, only in the boost namespace instead of in the std namespace as the standard specify.

One challenge when using POSIX threads in C++ programs is good ways to combine locking with exceptions. Locks in POSIX threads are not scoped, which means that every exception between acquiring a lock and its release has to be caught, followed by code to release acquired locks, followed by re-throwing the exception. Listing 4.3 shows this with C++ code, while Listing 4.4 shows how much easier this is with the new thread API.

Listing 4.3: Pthread example

```
1 void example(int protected_int) {
```

```
2     try {
3         pthread_mutex_lock(&mutex);
4         protected_int += 1;
5     } catch(...) {
6         pthread_mutex_unlock(&mutex);
7         throw;
8     }
9     pthread_mutex_unlock(&mutex);
10    return protected_int;
11 }
```

Listing 4.4: Boost thread example

```
1 void example(int protected_int) {
2     boost::lock_guard<boost::mutex> lock(mutex);
3     return protected_int;
4 }
```

Another challenge with POSIX threads is sending arguments into a thread. With POSIX threads, one can only send a single void pointer. C++0x makes this easier by allowing an arbitrary number of arguments of any copyable type.

POSIX threads is a far more complete and well tested API than C++0x threads. However, the discussed improvements with C++0x threads over POSIX threads simplify our programs to such an extent that we consider this a risk worth taking.

4.5.7 Naming

There is much research that shows the importance of good naming [60–63]. In *P2G-RPC*, we have a template-based thread-safe general purpose queue that is called `MsgQueue`. This should be given a name reflecting its true nature, such as `ThreadSafeQueue`. Our `SenderPool` is not really a pool, but a queue, thus `SenderQueue` would be a better name. The `RPCService` has nothing to do with RPC, and neither `RPCService` nor `IOService` are subclasses of the `Service`-class, and would be described better with names such as `ServiceQueue` and `IOEventHandler`.

4.5.8 Summary of issues

The issues we have discussed can be broken down into two groups.

1. Unneeded complexity. Not only in our code, but also in our library choices.
2. Code that is hard to test.

P2G-RPC did exactly what a prototype is supposed to do. However, as the P2G project developed further and as we worked on the implementation on *P2G-RPC*, we figured out, to a far greater extent, what we require. *P2G-RPC* has so many issues and bugs that we decided to create a new version from scratch. We needed to make it from scratch to overcome the issues with global data in Singletons and testing.

4.6 New design goals: Lessons learned

Creating a new version of the socket library allowed us to incorporate our solutions to all our issues with the initial design. Our goal was a minimalistic, modular, clean, well tested library which is easy to maintain and optimize.

4.6.1 Minimalistic

We made three design choices based on the goal of minimalism. Our first choices are based on the issues discussed above.

- All threads come from a single pool. This pool can be joined just like any single thread, and they can be asked to stop after completing their current job, thus allowing for *graceful shutdown*.
- We use the Boost thread API, since it provides us with an API that integrates with C++ in ways that reduce the complexity of all threading code.
- Our socket code is hidden behind an interface which can be implemented with any socket library. The current implementation uses Boost ASIO.

4.6.2 Modular, clean and easily maintainable

A clean modular design makes it easy to replace any part of a system. Continuing change is an inherent part of research, so a modular design makes sense. Our design was made with the following four general design principles in mind.

- Each function or method does exactly one thing, and it should be very short [63].
- Each class has exactly one responsibility [63].
- Each namespace represents exactly one layer of functionality.
- All names are self documenting.

The first three items in the list encourages modular code that is easy to test and maintain. Item 4 requires further explanation and justification.

Lessons learned from our central role in designing, maintaining and stabilizing the Devilry open source project [100], and ideas from the KDE project [101] and the Google C++ style guide [102], have shown that this rule should be used for all names, even internal variables. Consider a service receiving an event. The service is a method with the received event as one parameter. It might be tempting to name this parameter *e*, or *event*. The issue with naming arise when our service need to create an event of its own, and many choose to solve this with a new variable, *e2*, or even *x*. If we had given the parameter a proper name to begin with, such as *receivedEvent*, and our second event was given a good self documenting name such as *requestMoreResourcesEvent*, our code instantly would have become easier to read, harder to misunderstand, and thus easier to maintain.

4.6.3 Testing

Our communication library is designed to be a fundamental component of P2G, which has a large and steadily growing code base. Without tests ensuring that at least the fundamental parts of the library works, any integration into P2G would have to consider errors in both the new P2G code and in the communication library. Automatic tests are far more useful than tests that require any kind of user

interaction, since they can be run after every change in the library to ensure that new changes do not break any existing features. Our updated communication library has no global state, and minimal dependencies on other parts of the library, which enables us to test any independent part of the library in isolation from any other part of the library.

4.6.4 Terminology

The issues with naming in the initial design, and the unclear definition of the RPC terminology, made us reconsider our terminology. We realized that asynchronous message passing is basically the same as event handling. Our listeners behave like event listeners, and messages have some unique property identifying its purpose, just like events. This research made us realize that our library could be designed to handle events in a transport independent manner, thus enabling event listeners both within the same address space and on a different computer in the network. Such a design enables us to design applications in terms of services without considering their location. Location of services can be deferred to the optimization stage, and it can be adaptive over time. This complements P2Gs design, where tightly coupled processes are migrated as close to each other as possible.

4.7 *Sevent* - A socket event library

Changing terminology, coupled with a rewrite from scratch made it natural to choose a name for the library reflecting these changes. We named it Socket Event (*Sevent*), reflecting the use of the event terminology, and sockets for communication. The library is split into the four modules shown in Figure 4.2. Each module has minimal dependencies on the other modules, and their dependencies are clearly defined, making it possible to replace each module. The socket module takes care of all network communication, the serialize module contains utilities for serialization for various data types, and the event module implements event handling on top of the socket module. The datastruct module has no depen-

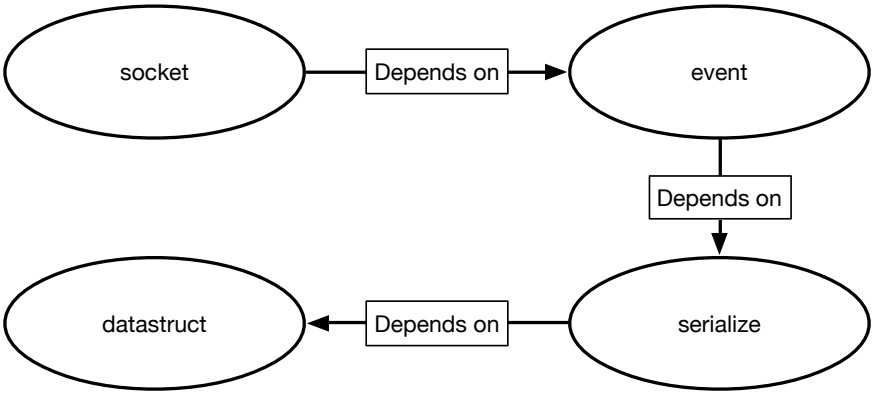

Figure 4.2: *Sevent* modules.

dencies on any other part of the library, it only contains general purpose data structures required by other parts of the library.

4.7.1 serialize - The serialization module

The *serialize* module is a framework which enables delayed serialization of many data types, and an API which makes it easy to add support for any data type.

Delayed serialization means that the framework enables us to delay serialization until it is required, such as when communicating the data via a socket. In P2G, tasks often run on the same machine. It would be inefficient for such local tasks to serialize their communications. Delayed serialization enables our event module to only serialize communications with other computers on the network. To achieve delayed serialization, we couple our data with a pair of serialization and deserialization functions, and when we need to serialize or deserialize our data, we use this pair.

With delayed serialization, the sender and the receiver have no way of knowing if the data originated locally, or on another machine. With local transfer, a pointer to the data is given directly to the receiver without any changes. With transfer via a remote connection, the receiver will get a pointer to deserialized data. To avoid memory leaks, we can free the local data after serializing it. However, the sender may need to continue using the data after sending it. If we free the data automatically, the sender would have to copy the data before sending it. If we

do not free the data automatically, the sender and receiver will have to know if the data was transferred locally or via a remote connection, and handle each case differently to avoid memory leaks. With local transfer, the receiver could free the data, but only after making sure that the sender is done using it. With transfer via a remote connection, the sender can free the data at any time after it has been sent.

A solution to this issue is to use a *smart pointer* [64] implementation such as the *shared_ptr* type [103] in the Boost library. *shared_ptr* objects store a pointer to a dynamically allocated object, and guarantees that the object is deallocated when the last *shared_ptr* pointing to the object is destroyed. If all data to be transferred are smart pointers, the data will be deallocated as soon as it goes out of scope. For local transfers, the smart pointer will be copied to the receiver, and when the smart pointer goes out of scope on both sender and receiver, it will be deallocated. For transfer via a remote connection, the sender will treat the smart pointer exactly as for local transfers. However, the receiver will receive a smart pointer to data allocated by the deserialize-function, and this data will be deallocated when the smart pointer goes out of scope on the receiver.

One issue with using smart pointers as described, is that all data must be smart pointers. Some data might be statically allocated on the sender, which means that we have to copy the data into a dynamically allocated data structure before sending the data. We work around this through the use of another data type from the Boost library, namely *Boost.Any* [104]. Boost.Any can store any copy constructable data type. *shared_ptr* is copy constructible, so they continue to work as previously described. However, using Boost.Any allows us to send other forms of intelligent pointers. We can, for example, wrap a pointer to a statically allocated array in our own copy constructible data type, which we will call `StaticArray`. With local transfer, our data type is copied, and the pointer can be retrieved by the receiver with zero overhead. With transfer via a remote connection, the serialize function can serialize the array just as it would do with a shared pointer. The remote receiver can store the deserialized received data as a smart pointer, and `StaticArray` can use this data. `StaticArray` will have to contain both a normal pointer and a shared pointer, and only use one of them at any given time. However, the overhead of this is minimal.

Boost serialization

Boost serialization is, as mentioned in Section 4.2, able to serialize most C++ data types, including complex data structures such as vectors, maps and class hierarchies. Serialization with Boost has some overhead, but its support for more or less any data type, and its ease of use, makes it a very good method of serialization when performance is not essential or when the data structures are so complex that no other alternative exists. Therefore, one of the serialization techniques supported by the serialize module is Boost serialization.

Integer arrays

Since Boost serialize has some overhead, we need more efficient methods for transfer of the most commonly used data types by store and fetch statements in P2G kernels. Among the current workloads targeted by P2G, a vast majority of fetch and store data is arrays of integer data. To minimize processing overhead, our implementation sends integer arrays without any changes to the data. As long as the target system uses the same endianess to represent integers in memory, this works. However, between computers using different endianess, we need to change endianess when deserializing the array. Since most computers in our *target global topology* use the same endianess, this is very efficient, and we only transfer a single additional byte with each array to indicate the edianess with which it was sent.

4.7.2 event - The event handling module

In *Sevent*, an event is a unique identifier followed by its payload, which is a list of pairs of size and data. Events do not know anything about the format of the data, leaving that to the serialization module. This leaves us with a flexible framework witch enables us to use the most suitable serialization technique for each pair in the list. This structure makes it easy to send complex series of data such as a serialized class, followed by several arrays of different types.

Listing 4.5: SerializablePerson.h - A class which is serializable using Boost serialization

```
#pragma once
#include <boost/shared_ptr.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/access.hpp>

/** A serializable person */
class Person {
    public:
        unsigned short age;
        std::string name;
    public:
        Person() {}
        Person(const std::string& name_,
                unsigned short age_) :
            name(name_), age(age_) {}

    private:
        friend class boost::serialization::access;
        template<class Archive>
        void serialize(Archive& ar,
                       const unsigned int version) {
            ar & age & name;
        }
};
typedef boost::shared_ptr<Person> Person_ptr;
```

Listing 4.6: *Sevent* event module example

```
#include <boost/shared_ptr.hpp>
#include <boost/make_shared.hpp>
#include <iostream>
#include <string>
#include "sevent/sevent.h"
#include "SerializablePerson.h"

using namespace sevent;
using namespace sevent::event;
typedef boost::shared_ptr<std::string> String_ptr;

int main(int argc, const char *argv[]) {
```

```
13      String_ptr helloIn, helloOut;
14      Person_ptr supermanIn, supermanOut;
15
16      helloIn = boost::make_shared<std::string>("Hello");
17      supermanIn = boost::make_shared<Person>("Superman", 39);
18
19      Event_ptr event = Event::make("example::MyEvent");
20      event->push_back(Buffer::make(helloIn,
21                                    serialize::String));
22      event->push_back(Buffer::make(supermanIn,
23                                    serialize::Boost<Person>()));
24
25      helloOut = event->first<String_ptr>(serialize::String);
26      supermanOut = event->at<Person_ptr>(1,
27                                    serialize::Boost<Person>());
28      std::cout << *helloOut << std::endl;
29      std::cout << supermanOut->name << std::endl;
30      return 0;
31  }
```

Events are stored in objects of the `sevent::event::Event` type. Listing 4.6 shows how the event API is used by the programmer. On line 19, we create a new event, with *"example::MyEvent"* as identifier. Next we add a string and an object of the Person class from Figure 4.5. Notice how the buffer class requires a pair of data and serializer as explained in Section 4.7.1. At this point, we have created an event which can be serialized and sent to another computer, or passed around within the current process.

The event-object has a method, `serialize_at`, which takes an index as argument and returns a serialized version of the buffer at that index. It also has a method, *size*, which returns the number of buffers in the event. These two functions together is what our socket module requires to be able to iterate over an event and serialize all its buffers.

Lines 25-26 show how we retrieve data from our event. The event object has a shortcut for retrieving its first buffer, and any buffer can be retrieved using the *at*-method. Just as with the push_back method, we need to supply the serializer to enable the event object to deserialize the data before returning it if required. In this example, deserialization is not required, since we defer serialization to the

socket module, and we never send our event to the socket module.

Event identifiers

Listing 4.6 shows that event identifiers are strings. Our initial implementation used integers as identifiers. However, the first attempts at integration of the *Sevent* library into P2G revealed that numeric identifiers are hard to maintain in a development environment with multiple concurrent contributors, since all the developers have to cooperate to create unique identifiers. One solution is to use a global table of identifiers. However, this tightly couples any component to this table. In P2G, this is not even possible, since some components, such as *Olib*, are developed outside the P2G code base, and thus, choose its own event ids. On a research project such as P2G, one will continually get contributions from multiple sources, and we already have a well established method of distinguishing these sources. This method is project naming. With a string as identifier, one can prefix every event id with the name of the project, such as *olib* and *instrumentation*.

Listing 4.7: *Sevent* StringEventId

```
1  #pragma once
2  #include <stdint.h>
3  #include <string>
4  #include "EventIdBodySerialized.h"
5
6  namespace sevent {
7  namespace event {
8
9  class StringEventId
10 {
11 public:
12     typedef std::string value_type;
13     typedef const value_type& value_typeref;
14     typedef uint8_t header_type;
15     typedef uint8_t header_network_type;
16 public:
17     static StringEventId_ptr makeFromNetwork(
18                             header_type header, char* body) {
19         return boost::make_shared<StringEventId>(
20                                         std::string(body));
```

```
21      }
22
23      static unsigned headerSerializedSize() {
24          return 1;
25      }
26
27      static header_type deserializeHeader(
28                                  header_network_type header) {
29          return header;
30      }
31
32      static uint8_t bodySerializedSize(header_type header) {
33          return header;
34      }
35
36      static bool hasBody() {
37          return true;
38      }
39
40  public:
41      StringEventId(value_typeref value) :
42          _value(value) {}
43
44      header_network_type serializeHeader() {
45          return static_cast<uint8_t>(_value.size()+1);
46      }
47
48      EventIdBodySerialized serializeBody() {
49          return EventIdBodySerialized(_value.size()+1,
50                                       _value.c_str());
51      }
52
53      const value_type& value() {
54          return _value;
55      }
56
57  private:
58      value_type _value;
59  };
60
61  } // namespace event
62  } // namespace sevent
```

Using strings as identifiers have some overhead. Instead of sending a single integer, which is 4 bytes, we send a single byte containing the length of our identifier, followed by the identifier. Currently, no events in P2G are so small that this overhead is noticeable, however, in case this might become an issue, we chose to make the identifier type easy to redefine. We achieve this by defining a event identifier interface where it is possible to define identifiers consisting of a fixed size header, followed by a body. Numeric identifiers use only the fixed size header, and the socket module ignores its body since it is empty. String identifiers use the length of the string as header, and the identifier as body. With this approach, it is easy to add other types of identifiers. Listing 4.7 shows how string events are implemented. We can see that it provides a method to determine its header size, which is 1, and methods to determine if it has a body and the size of the body. Both header and body can be serialized, which enables us to use any data type as both header and body.

The issue of strings taking much space might be worked around using hashing [65]. The idea behind string hashing is to filter the string through a hashing algorithm which returns a numeric value which is inexpensive store and compare. However, any algorithm turning a long string into a small number will produce the same number for different strings. Even if we choose a numeric data type capable of containing huge numbers, such as a 164 bit integer, collisions are still possible. Furthermore, a 164 bit number would consume the same amount of space as a string identifier of 19 characters, so we would not really gain anything through such an approach. We conclude that hashing can not be used to implement string identifiers in our framework.

Another approach is packing strings with a character set consuming less than 8 bits. We could make a character set supporting only lower case English characters and some additional characters, such as whitespace or other suitable separators. Since the English alphabet contains 26 numbers, we could use 5 bits for each character. This saves some space, and could be considered in the future if the overhead of strings as event identifiers causes performance issues.

Event handling

When events are sent to the socket module, they are serialized unless the receiver is within the same process as the sender. On the receiver, events are received by a worker thread, and forwarded to a single event handler, dubbed *all-events-handler*, which runs within the worker thread. This handler is defined by the programmer using the library. A very common use case is to have a handler for each kind of event. We support this as a separate submodule, which can be used within the *all-events-handler*. This submodule also contains a simple *all-events-handler*, which does error handling suitable for debugging.

The prime reasons for choosing the *all-events-handler* approach instead of hard coding a system where each event is forwarded to a specific handler, is modularity and error handling. We wish to split this code into separate modules to enable fine tuned optimization of our event handler delegation. All event handlers are created by the programmer using the library. Leaving the handling of these events to the programmer, enables the programmer to handle his own errors without any confusing abstractions. Since worker threads run event handlers, we risk starvation when all worker threads are waiting for an event to proceed. Our modular approach makes this an issue for a module built on top of the *all-events-handler*. We could solve this by spawning a new thread for each incoming event, but that would be inefficient since we would have no way of limiting the number of incoming events. Using a pool of threads would just move the issue of starvation to the pool. By giving the programmer using the library full control over event handling, we allow the programmer to prevent starvation in an efficient way tailored to the flow of events in each program.

4.7.3 socket - The network communication module

Our network communication module API (Figure 4.3) is far less complex than our first design.

Listener is the interface used to accept new connections.

Connector is the interface used to create new connections to a listener. Estab-

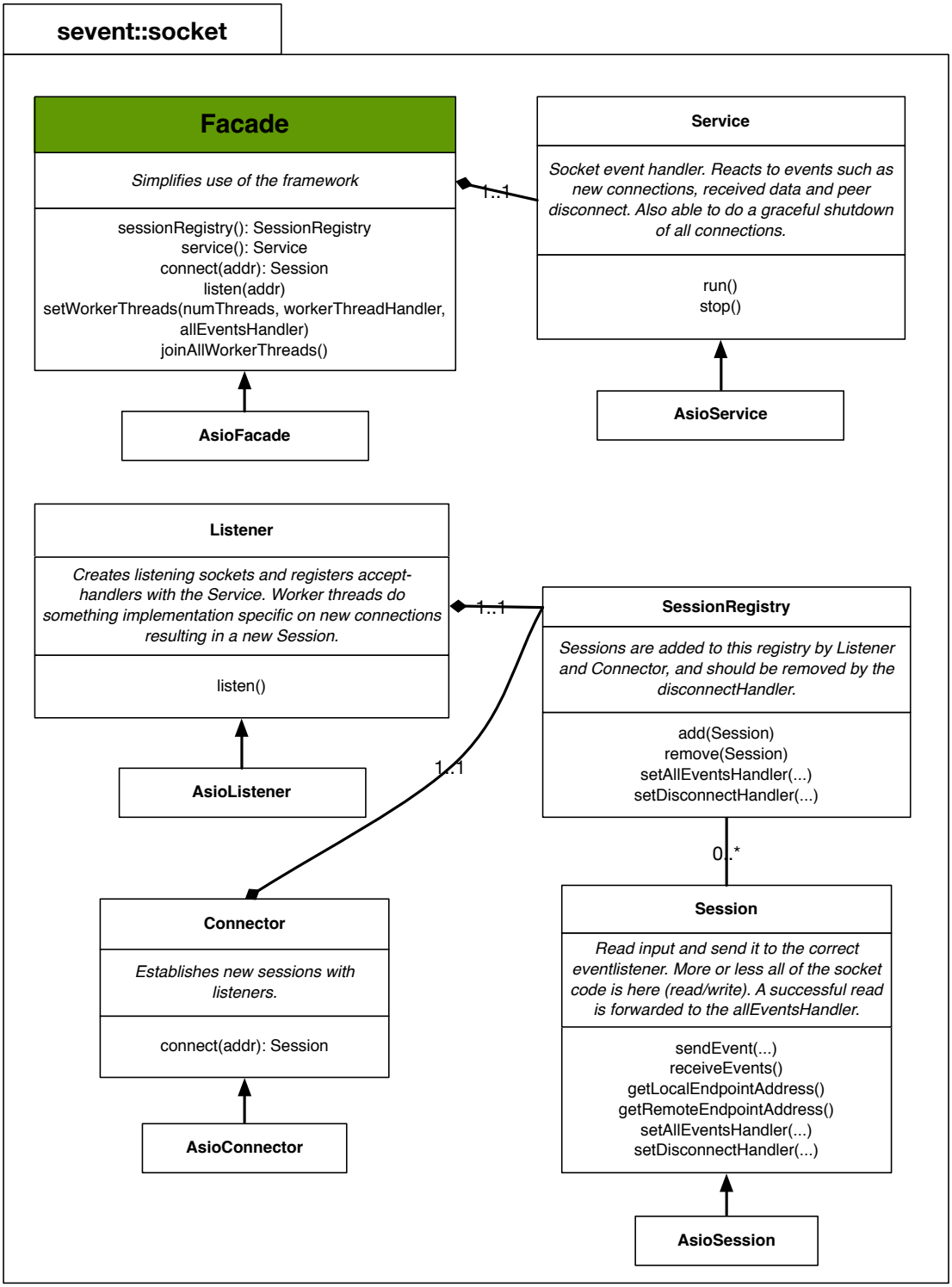

Figure 4.3: UML diagram of the socket layer in *Sevent*.

lished connections are stored in `SessionRegistry`.

SessionRegistry contains all active sessions. A `Session`-object can receive and

send events. Worker threads react to incoming events and new connections, and forward them to the receivedEvents-method in `Session` or to a `Listener`.

Facade binds the socket layer of the library into an easy to use API. The facade uses a `Service` object to start and stop event handling.

The module is an interface, and we provide a TCP-based implementation using Boost Asio. However, adding an implementation using a more sophisticated application layer protocol, as we discussed in Section 4.3, would only require an additional implementation of Listener, Connector and Session.

4.7.4 Example

Listing 4.8 and Listing 4.9 shows how *Sevent* work in practice. In this example, the client sends 4 events.

"example::Msg" contains a single string.

"example::Person" contains a person object.

"example::Array" contains an array of 32 bit unsigned integers.

"example::Die" is an event without any data.

To keep the example simple, each event has a single data buffer. Events with multiple data buffers were explained in Section 4.7.2.

When we send these events using the `sendEvent` method of the facade object, the facade checks if the given session was created within the facade, detects that it is not so, serializes the event and sends it to the server. If we were to register the event handlers in the server with the facade used by the client, and use a session with this facade, `sendEvent` would transfer the events directly to the event handlers without serialization.

Listing 4.8: *Sevent* example client

```
1 #include <boost/lexical_cast.hpp>
2 #include "sevent/sevent.h"
3 #include "SerializablePerson.h"
```

```
4
5  using namespace sevent;
6  using namespace sevent::socket;
7  using namespace sevent::event;
8  using namespace sevent::datastruct;
9
10 int main(int argc, const char *argv[]) {
11     if(argc != 2) {
12         std::cout << "Usage: " << argv[0]
13             << " <server-addr>" << std::endl;
14         return 1;
15     }
16     std::string serverAddr(argv[1]);
17
18     Facade_ptr facade = Facade::make();
19     Session_ptr session = facade->connect(Address::make(serverAddr))
           ;
20
21     // Event data
22     typedef boost::shared_ptr<std::string> String_ptr;
23     String_ptr hello = boost::make_shared<std::string>(
24                                         "Hello");
25     Person_ptr superman = boost::make_shared<Person>(
26                                         "Superman", 39);
27     boost::shared_array<uint32_t> array = boost::shared_array<
           uint32_t>(new uint32_t[3]);
28     array[0] = 10;
29     array[1] = 20;
30     array[2] = 30;
31     Uint32SharedArray_ptr arrayContainer = boost::make_shared<
           Uint32SharedArray>(array, 3);
32
33     // The events
34     Event_ptr helloEvent = Event::make("example::Msg",
35                 Buffer::make(hello,
36                              serialize::String));
37     Event_ptr supermanEvent = Event::make("example::Person",
38                 Buffer::make(superman,
39                              serialize::Boost<Person>()));
40     Event_ptr arrayEvent = Event::make("example::Array",
41                 Buffer::make(arrayContainer,
42                              serialize::Uint32SharedArray));
```

```
43
44      // Send the events
45      facade->sendEvent(session, helloEvent);
46      facade->sendEvent(session, supermanEvent);
47      facade->sendEvent(session, arrayEvent);
48      facade->sendEvent(Address::make(serverAddr), helloEvent);
49      facade->sendEvent(session, Event::make("example::Die"));
50      return 0;
51  }
```

Listing 4.9: *Sevent* example server

```
1  #include <iostream>
2  #include <boost/bind.hpp>
3  #include <boost/lexical_cast.hpp>
4  #include "sevent/sevent.h"
5  #include "SerializablePerson.h"
6
7  using namespace sevent;
8  using namespace sevent::socket;
9  using namespace sevent::event;
10 using namespace sevent::datastruct;
11
12 void helloHandler(Facade_ptr facade,
13                   Session_ptr session,
14                   Event_ptr event) {
15     typedef boost::shared_ptr<std::string> String_ptr;
16     String_ptr data = event->first<String_ptr>(
17                               serialize::String);
18     std::cout << "EventId=" << event->eventid()
19         << "  Data=" << *data << std::endl;
20 }
21
22 void personHandler(Facade_ptr facade,
23                    Session_ptr session,
24                    Event_ptr event) {
25     Person_ptr p = event->first<Person_ptr>(
26                               serialize::Boost<Person>());
27     std::cout << "Person-event received! "
28         << p->name << ":" << p->age << std::endl;
29 }
30
```

```
31 void uint32ArrayHandler(Facade_ptr facade,
32                         Session_ptr session,
33                         Event_ptr event) {
34     Uint32SharedArray_ptr arrayContainer = event->first<
           Uint32SharedArray_ptr>(serialize::Uint32SharedArray);
35     boost::shared_array<uint32_t> array = arrayContainer->array();
36     std::cout << "Shared array-event received! " << std::endl;
37     for(int i = 0; i < arrayContainer->size(); i++) {
38         std::cout << "sharedArray[" << i << "] = "
39             << array[i] << std::endl;
40     }
41 }
42
43 void dieHandler(Facade_ptr facade,
44                 Session_ptr session,
45                 Event_ptr event) {
46     std::cout << "*** DIE-event received ***" << std::endl;
47     facade->service()->stop();
48 }
49
50 int main(int argc, const char *argv[]) {
51     if(argc != 2) {
52         std::cout << "Usage: " << argv[0]
53             << " <address>" << std::endl;
54         return 1;
55     }
56     std::string address(argv[1]);
57
58     Facade_ptr facade = Facade::make();
59
60     // Setup the eventhandlers
61     HandlerMap_ptr eventHandlerMap = HandlerMap::make();
62     eventHandlerMap->addEventHandler("example::Msg",
63                                      helloHandler);
64     eventHandlerMap->addEventHandler("example::Person",
65                                      personHandler);
66     eventHandlerMap->addEventHandler("example::Array",
67                                      uint32ArrayHandler);
68     eventHandlerMap->addEventHandler("example::Die",
69                                      dieHandler);
70
71     // Start 5 worker threads,
```

```
72      // and use the handler above for incoming events.
73      facade->setWorkerThreads(5,
74              boost::bind(simpleAllEventsHandler,
75                          eventHandlerMap,
76                          _1, _2, _3));
77
78      // Create a listening socket.
79      facade->listen(Address::make(address));
80
81      // Wait for all work to finish. In this example this will
82      // happen when the dieHandler calls
83      // facade->service()->stop().
84      facade->joinAllWorkerThreads();
85
86      return 0;
87  }
```

The server contains event handlers for each event, `helloHandler`, `personHandler`, `uint32ArrayHandler` and `dieHandler`. These handlers are registered with the facade in the `main`-method, and triggered when the events are received. Each of the handlers, except for `dieHandler`, prints the received data to standard output. The handlers could use their `facade` and `session` arguments to send events back to the client, but this would require event handlers on the client, which we excluded from the examples to keep them simple. Handlers on the client would work exactly like they do on the server, and they would be registered in exactly the same way. The `dieHandler` tells the service to stop handling events, which makes the `joinAllWorkerThreads`-method of the `facade`-object called at the end of the main function return, effectively stopping the server.

4.7.5 Automatic tests

Listing 4.10: Automatic tests with *Sevent*

```
1 #define BOOST_TEST_DYN_LINK
2 #define BOOST_TEST_MODULE TestExample
3 #include <boost/test/unit_test.hpp>
4 #include "sevent/sevent.h"
```

```
5
6  using namespace sevent;
7  using sevent::event::Event;
8
9  unsigned counter = 0;
10 void allEventsHandler(socket::Facade_ptr facade,
11                        socket::Session_ptr session,
12                        event::Event_ptr event) {
13     counter ++;
14     if(counter == 2) {
15         facade->service()->stop();
16     }
17 }
18
19 BOOST_AUTO_TEST_CASE(SimpleTestExample) {
20     socket::Address_ptr listenAddr = socket::Address::make(
21                                             "127.0.0.1", 9091);
22     socket::Facade_ptr facade = socket::Facade::make();
23     facade->setWorkerThreads(1, allEventsHandler);
24     facade->listen(listenAddr);
25     socket::Session_ptr session = facade->connect(listenAddr);
26
27     facade->sendEvent(session, Event::make("ping"));
28     facade->sendEvent(session, Event::make("ping"));
29
30     facade->joinAllWorkerThreads();
31     BOOST_REQUIRE_EQUAL(counter, 2);
32 }
```

As explained in Section 4.6.3, the ability to write automatic tests embedding *Sevent* is an important property of the library when it is to be integrated into multiple layers of the P2G code base. Listing 4.10 is a minimal example of a unit test written for the Boost test library [105]. We define a custom *all-events-handler*, which counts how many incoming events it receives, and stops the service from handling more events when this counter reaches two. In the body of BOOST_AUTO_TEST_CASE(SimpleTestExample) starting on line 19, we set up a facade, create a connection to this facade, send two events, and wait for them to be handled by the event handler. When the event handler stops the service from handling more events, we assert that counter equals 2. This is about as simple as

such a test can be, however it is a demonstration of the viability of writing automatic tests using *Sevent*. Further proof of the usability of *Sevent* in automatic tests can be found in the *test/socket/* directory in the *Sevent* source code, which contains tests for the entire socket module.

4.8 Summary

Through several iterations of design and testing of *Sevent*, we have made a library that can be used for all network communication in P2G. Our primary experience from this research is that simple and minimal external APIs in each module in a subsystem, makes it easy to integrate a subsystem into a larger system, such as P2G.

Furthermore, automatically testing as much of the code as possible revealed errors at an early stage. Detecting most errors early made it very easy to integrate the various submodules, since most errors related to such integration was due to bugs in the way the submodules integrated instead of bugs in each submodules. Tracking down these bugs was relatively easy, since a quick look at our automatic tests for each submodule eliminated most of the possible sources for each bug.

Another important experience is the importance of good naming and terminology when integrating with modules developed by multiple developers. Misunderstandings are far less frequent if the terminology is solid, and descriptive and unambiguous names are used.

Our next challenge is to integrate *Sevent* with *Olib*. This should ease the integration of network distribution into *Olib*. Successful integration with *Olib* would be a good indication of the usefulness of *Sevent*. Furthermore, the integration of these two libraries can serve as a test of the quality of the external APIs in both libraries, especially *Sevent*, since its API should work without further changes.

Chapter 5

Network distribution of Obese libraries

In Chapter 3, we introduced the ideas behind *Olib*, and how to implement it on a single heterogeneous computer. The *Sevent* library discussed in Chapter 4 is designed to meet our requirements for local and network communication. In this chapter, we demonstrate the improvements the *Sevent* library brings to our initial *Olib* implementation, and the changes required for a distributed version of *Olib*.

5.1 *Olib* on a single heterogeneous computer

Our initial implementation of single computer Obese libraries (Section 3.7) worked satisfactory, except for error prone and complex socket communication between the client application and the scheduler. Using *Sevent*, we can reduce communication complexity considerably. We can make all requests and responses into events.

When integrating *Sevent* with *Olib*, we defined all requests and responses as classes and serialized objects of these classes with Boost serialization. Since we used classes, it was easy to add and remove information to our events. Minimizing the amount of work required to change requests and responses saved much time during our research and during the migration to a distributed version. Serializing classes is not very efficient. However, these classes are simple,

so the overhead of serialization should be minimal. Furthermore, our focus is not scheduling, so we have not spent any time optimizing scheduler communication.

5.2 *Olib* distributed in a network of heterogeneous computers

Adapting the single computer *Olib* implementation for network distribution did not require many changes. The complexities of network communication is handled transparently by *Sevent*. The scheduler in our single computer implementation is a stand-alone process using sockets to communicate with the client applications. This means that the single computer implementation of *Olib* partly supports distribution. The scheduler and client application can be on different computers in the network.

What we are missing is information from the scheduler about where to load the shared libraries. In the single computer implementation, the *scheduling response* contains the location of the shared library that executes the kernel instance on the selected processing unit. The *scheduling response* does not contain any information about what computer to load the shared library on, since it is to be loaded by the client application process running on the same computer. If we add the network address of the computer where the P2G kernel instance is to be executed to the *scheduling response*, the client application can send a *run request* containing the location of the shared library to the computer at the given address. Computers eligible to receive *run requests* need to have a *run handler* that can load shared libraries. Computer executing a *run handler* is called a *slave*. We call them *slaves* since they have to do as they are told by the scheduler.

Figure 5.1 show how requests and responses flow between request and response handlers in the distributed *Olib* implementation, and Handler-listing 2 describes each handler in detail. Comparing these listings to the corresponding single computer implementation (Figure 3.1 and Handler-listing 1), we can see that the difference is minimal. The major difference is that we move the task of loading shared libraries from the client application into the slaves. The changes shifts the

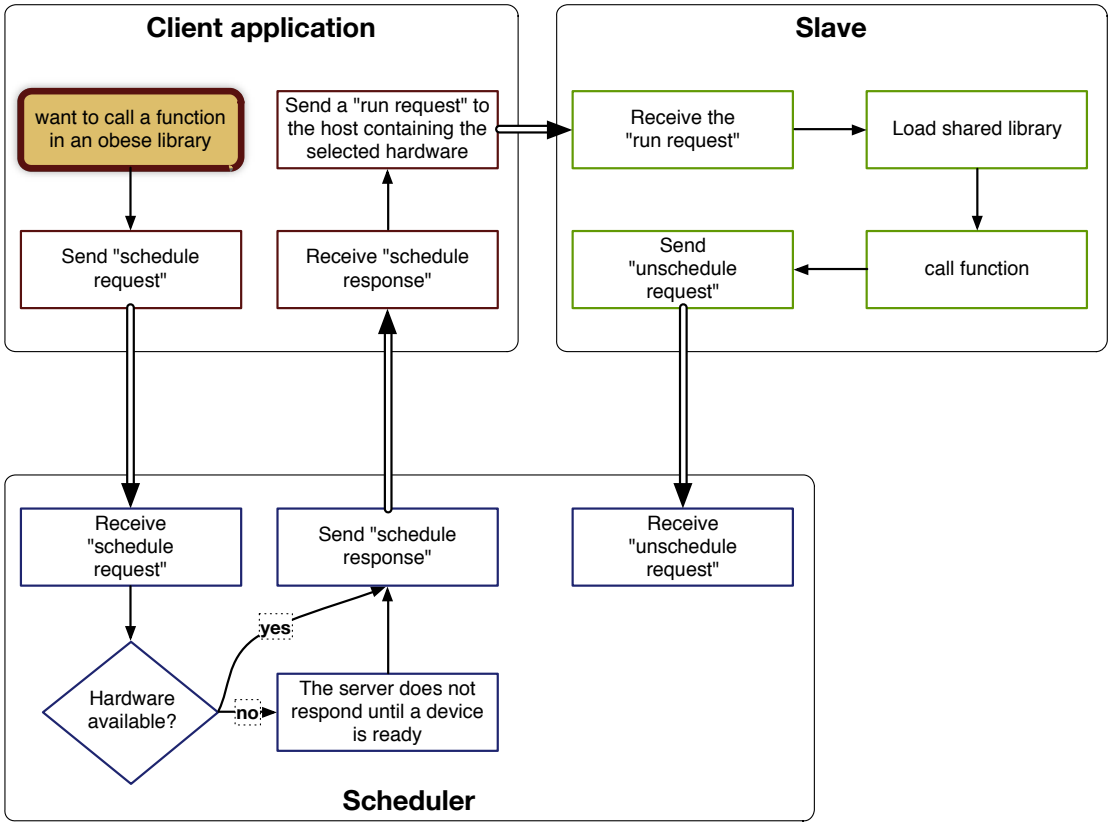

Figure 5.1: Flow of requests and responses in the distributed *Olib* implementation. Arrows with double lines represent network communication, while the other arrows represent internal communication within the same process.

code required by the P2G runtime from the client into the slave. We assume the P2G runtime will be a superset of our slaves, therefore we continue calling them slaves.

5.3 Issues with our design

The design outlined in the previous section works, however, it has a minor issue that can be solved by adding a bit of extra complexity. When a client application sends a *run request*, it does not get any response. This is because all communication in *Sevent* is asynchronous. Therefore the client can not know when the shared library has been loaded and the function has started executing, which means that the client can not initiate communications with the executing func-

Schedule request handler receives *scheduling requests* containing the file-system path to an application directory and the name of a function that every shared library within the application directory must provide. The scheduler responds with the file-system path, the name of a shared library located within the application directory, and the name of the function. The name of the shared library is selected by the scheduler from a list of shared library names and slave addresses in a round-robin manner. We assume that a scheduler integrated into P2G would use a more dynamic method of choosing this name using instrumentation data.

Schedule response handler receives responses to scheduling requests containing the address of the selected slave, and sends *run requests* to the *run handler* on this slave.

Run handler receives *run requests* containing the file-system path to the application directory, the name of a shared library within the application directory, and the name of a function within the shared library. The *run handler* loads the shared library and invokes the function in a separate thread. When the execution of the function completes, the scheduler is notified using an *unscheduling request*.

Unscheduling request handler receives *unscheduling requests* containing the address of the slave, the file-system path to an application directory, and the name of a shared library within the application directory. This is enough information for the scheduler to know that the processing unit is used by one less kernel instance.

Handler-listing 2: Handlers in distributed *Olib*. Changes from Handler-listing 1 are highlighted with larger font and green color.

tion. This means that any input data to the function has to be part of the *run request*, and there is no simple way of communicating data back to the client. The *Sevent* library makes two way event passing trivial, and supporting such communication makes the *Olib* library more flexible. For example, it makes it possible to send data, such as statistics, back to the client, while forwarding results to another slave.

In our design, there is much unneeded communication overhead. First of all, objects serialized with Boost using the stable plain text serialization format recommended for endian-safe data transfer consume more communication bandwidth than an optimized binary protocol and even more than a custom built plain text protocol. If this becomes an issue, *Sevent* makes it relatively easy to replace our se-

rialization method with a more space-optimized binary protocol such as Google protocol buffers or the less stable endian-safe binary protocol in Boost serialization.

The second form of communication overhead is with the protocol itself. To keep our handlers as independent as possible, we opted for a protocol where each handler receives all the data they need to independently complete their task. This means that the *schedule request handler* responds with all its input data in addition to the name of a shared library. The client only needs to get the name of the shared library back from the scheduler, since it already knows what it sent to the scheduler. However, since communication is asynchronous, such a solution requires the client application to maintain a registry of sent events waiting for response. Not only does this increase the complexity of the client application, it also binds the *schedule response handler*, and the module within the client responsible for sending requests more tightly, making testing of each component a more complex task. We have chosen to keep the client as simple as possible for our implementation, however optimizing communication should be considered in the future if profiling shows that it is a bottleneck.

5.4 Implementation details

Every request and response handler in *Olib* is implemented as *Sevent* event handlers. We have four classes that simplifies setting up and using these handlers for specific purposes. Each of these classes, shown in Figure 5.2 inherit basic functionality from `OlibBase`.

`OlibBase` contains a *Sevent* socket facade, and a *Sevent* event handler map.

`OlibMaster` adds the scheduling handlers to the event handler map. It is called master instead of scheduler, since it is a natural place to add handlers for other centralized P2G modules, such as an instrumentation handler.

`OlibClient` has the ability to connect to an `OlibMaster`, send scheduling requests and receive the response from the scheduler.

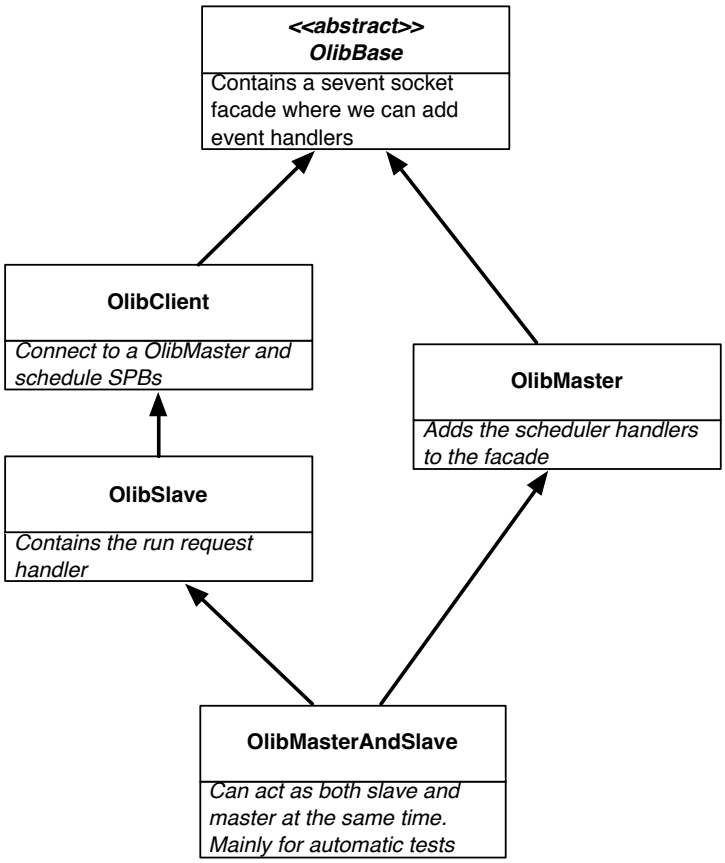

Figure 5.2: The class hierarchy of the *Olib* facades that makes it easy to create custom masters, slaves and clients.

`OlibSlave` extends `OlibClient` with the ability to handle *run requests* and load Obese libraries. It extends `OlibClient` because slaves can act as clients to other slaves, thus enabling us to forward data through any number of slaves without communicating back to the client.

`OlibMasterAndSlave` is a subclass of both `OlibMaster` and `OlibSlave`, which makes it possible to run scheduler, slave and client in the same process. This simplifies automatic testing of P2G applications.

Each of these classes make it possible to create schedulers, client applications and slaves with very few lines of code. This is very important for client applications, since every bit of added complexity with their common setup will increase the complexity of any client application. Slaves and schedulers are supposed to be implemented once and used anywhere. However with a clean and minimal API such as the one we get with our selected structure, automatic tests are easy to set

up, a feature we will explore further in Section 5.6.

5.4.1 Master

Listing 5.1: Source code for a *Olib* master

```
1  #include <boost/bind.hpp>
2  #include "olib/olib.h"
3  #include "olib/scheduler/Queue.h"
4
5  using sevent::socket::Address;
6  using olib::scheduler::Queue;
7  using olib::OlibMaster;
8  using olib::OlibMaster_ptr;
9
10 int main(int argc, const char *argv[]) {
11     Queue sched;
12     sched.add(Address("127.0.0.1", 3000), "x86");
13
14     OlibMaster_ptr master = OlibMaster::make(50,
15             boost::bind(&Queue::schedule, &sched, _1, _2),
16             boost::bind(&Queue::unSchedule, &sched, _1));
17     master->listen(Address::make("0.0.0.0", 9090));
18     master->socketFacade()->joinAllWorkerThreads();
19     return 0;
20 }
```

Our master uses a round-robin scheduler where we define a list of pairs of network addresses and shared library names that will be loaded in the given order. We name all shared libraries in our tests with appropriate names, such as *cuda*, *opencl* or *cpu*. Thus our scheduling list becomes a list of addresses and target processing unit. Each *scheduling request* removes an item from the list, and we add it back again upon receiving the corresponding *unscheduling request*.

Listing 5.1 is a complete scheduler application with a static schedule. The `Queue` object names *sched* defines our scheduler. We add a single address and shared library name to the scheduler, and bind the scheduler to a OlibMaster object. The *Olib* master listens on port 9090 on all interfaces, and waits for all worker threads

to finish. We have not added any means of stopping the worker threads, so this application will continue handling scheduling requests until it is killed.

5.4.2 Slave

Listing 5.2: Source code for a *Olib* slave

```
1  #include <boost/lexical_cast.hpp>
2  #include <boost/shared_ptr.hpp>
3  #include <boost/make_shared.hpp>
4  #include "olib/olib.h"
5
6  using sevent::socket::Address;
7  using olib::OlibSlave;
8  using olib::OlibSlave_ptr;
9
10 int main(int argc, const char *argv[]) {
11     if(argc != 3) {
12         std::cout << "Usage: " << argv[0]
13             << " <slave-addr> <master-addr>" << std::endl;
14         return 1;
15     }
16     std::string slaveAddr(argv[1]);
17     std::string masterAddr(argv[2]);
18
19     OlibSlave_ptr slave = OlibSlave::make(30);
20     slave->listen(Address::make(slaveAddr));
21     slave->connectToMaster(
22                 Address::make(masterAddr));
23     slave->socketFacade()->joinAllWorkerThreads();
24     return 0;
25 }
```

Listing 5.2 shows how our slave is implemented. The application takes its own and the masters address as input. The last five lines before the `main`-method returns is everything required for a fully functional *Olib* slave. It is initialized with 30 worker threads, starts listening on the given address and port, creates a connection to the master, and waits for all worker threads to finish. Just as with the scheduler, slaves never quit unless we kill them.

Since we have implemented a fully static scheduler, we have to add all slaves in

our network to the scheduler manually, and recompile the scheduler. We can set the number of concurrent instances of the same shared library allowed to run on a slave by adding exactly that many pairs of the same address and shared library name to the scheduler. For example, making three copies of line 12 in Listing 5.1 would permit four instances of a shared library named *x86* to run concurrently on the slave listening on *127.0.0.1:3000*.

Recompiling the scheduler for scheduling changes makes it time consuming to try out different schedules. Therefore, we have overloaded the constructor of `olib::scheduler::Queue` with a method taking a single string as input. This string specifies one or more pairs of slave addresses and shared library names, which makes it easy to update our master with an argument containing the schedule, thus making it possible to specify the schedule at runtime.

Loading shared libraries

The example in Listing 5.2 only shows a very high level API for creating slaves. Behind this high level API, the slave contains a *run handler*. As previously explained, the *run handler* receives *run requests*. Each run request makes the *run handler* load a shared library and invokes the requested function within the shared library.

The function is invoked with a single argument, a `Context`-object. The `Context`-object has information about the shared library, such as its location and name, a method for registering handlers for incoming events from clients, and a method for unscheduling the shared library.

We implement two-way communication as explained in Section 5.3. However, we do not implement data passing along with the *run request*. We chose this solution because it simplifies our implementation, however it does lead to some delays in communication as the sender must wait for a response before sending any data. A more optimal solution would be to support both forms of data transfer. The `Context` provides the `notifyInvoker`-method to notify the client that the function is ready to handle incoming data. This can not be done automatically since the shared library must manually add its event handlers for incoming data before it is safe to send data.

A normal method of processing multimedia workloads is a pipeline. Data is

passed through several algorithms, each algorithm processing input data and outputting data which is forwarded to the next algorithm in the pipeline. We support this through the `scheduleFunction`-method of the `Context` that results in a *schedule request*. The method takes the input required by the *schedule request handler*, and a callback that is invoked when the scheduled function notifies that it is ready to receive data. With this method, we can schedule a P2G kernel instance from a client application, and the kernel instance can schedule other kernel instances, thus pipelining data through an unlimited number of kernel instances.

5.4.3 Client

A client application works much like the slave. It needs to listen on a port if it is to be able to receive events from slaves. Just as the slave, it needs to create a connection to the master, however unlike the slave, it does not have to run until killed.

A slave gets a context from the run handler, which it uses to set up event handlers and schedule kernel instances. Clients also use contexts, however they have to create them manually. The contexts used by a client is almost like the context given as parameter to the function within the shared libraries, but since it is not within a P2G application, it does not contain the location and name of the current shared library. Using contexts on both clients and slaves provides the programmer with a unified interface for all scheduling and communication using our library.

5.5 A complete example

We have shown how the master and slave application are implemented, however we have only explained the inner workings of clients and slaves, and not how applications are implemented. In this section we demonstrate how to implement a distributed P2G application which takes two equal sized arrays as input, and produces an array containing the sum of the two input arrays. Our example was chosen for its simplicity, and not to demonstrate every single feature of *Olib*. This

is the low-level code we assume the P2G kernel language compiler will be able to produce for every target architecture.

5.5.1 The client application

We split our client applications over two files. This is mainly to enable us to use the client handlers in automatic tests. The client starts in Listing 5.3 by creating the input arrays. Next, it does what every client needs to do. It starts listening for incoming events and connects to the master. On lines 27-34, the client creates a context, adds an event handler listening for the result of the calculation, and schedules the `sumArrays`-olib-function with the `onInitSlaveConnection`-function, which is registered to be called when the olib-function signals that it is ready to receive input data. On line 37, the client blocks until the `onArraySumResult`-function signals that the result has been received, and the next lines print the results to standard output.

Listing 5.3: Client application calculating the sum of two arrays

```
1  #include <boost/lexical_cast.hpp>
2  #include <boost/shared_array.hpp>
3  #include <boost/bind.hpp>
4  #include "olib/olib.h"
5  #include "sumArraysClientHandlers.h"
6
7  using sevent::socket::Address;
8  using sevent::socket::Address_ptr;
9  using namespace olib;
10
11 int main(int argc, const char *argv[])
12 {
13     unsigned arraySize = 10;
14     boost::shared_array<uint32_t> result;
15
16     // Create the input arrays
17     boost::shared_array<uint32_t> arrayA = boost::shared_array<
           uint32_t>(new uint32_t[arraySize]);
18     boost::shared_array<uint32_t> arrayB = boost::shared_array<
           uint32_t>(new uint32_t[arraySize]);
19     for(int i = 0; i < arraySize; i++) {
```

```
20          arrayA[i] = i;
21          arrayB[i] = i+1;
22      }
23
24      OlibClient_ptr client = OlibClient::make(5);
25      client->listen(Address::make("127.0.0.1", 8000));
26      client->connectToMaster(Address::make("127.0.0.1", 9090));
27      protocol::Context_ptr context = client->createContext();
28      context->addEventHandler("example::ArraySumResult",
29              boost::bind(onArraySumResult, client, &result,
30                          _1, _2, _3));
31      context->scheduleFunction(
32                "sumArrays/sumArrays.olib", "sumArrays",
33                boost::bind(onInitSlaveConnection, _1,
34                            arrayA, arrayB, arraySize));
35      client->socketFacade()->joinAllWorkerThreads();
36
37      std::cout << "Array sum result received" << std::endl;
38      for(int i = 0; i < arraySize; i++) {
39          std::cout << "result[" << i << "] = "
40              << result[i] << std::endl;
41      }
42
43      return 0;
44  }
```

Listing 5.4: Implementation of sumArraysClientHandlers.h

```
1  #include "sumArraysClientHandlers.h"
2
3  using namespace sevent;
4  using namespace olib;
5  using namespace olib::protocol;
6  using namespace sevent::event;
7  using sevent::datastruct::Uint32SharedArray;
8  using sevent::datastruct::Uint32SharedArray_ptr;
9
10 void onInitSlaveConnection(
11                 SlaveSession_ptr slaveSession,
12                 boost::shared_array<uint32_t> arrayA,
13                 boost::shared_array<uint32_t> arrayB,
14                 unsigned arraySize) {
```

```
15      std::cout << "Slave for 'sumArrays()' found. "
16          << "Sending arrays." << std::endl;
17      Uint32SharedArray_ptr arrContainerA = boost::make_shared<
            Uint32SharedArray>(arrayA, arraySize);
18      Uint32SharedArray_ptr arrContainerB = boost::make_shared<
            Uint32SharedArray>(arrayB, arraySize);
19
20      Event_ptr arrayEvent = Event::make("example::ArrayData");
21      arrayEvent->push_back(Buffer::make(
22              arrContainerA, serialize::Uint32SharedArray));
23      arrayEvent->push_back(Buffer::make(
24              arrContainerB, serialize::Uint32SharedArray));
25      slaveSession->sendEvent(arrayEvent);
26  }
27
28  void onArraySumResult(OlibClient_ptr olibClient,
29                        boost::shared_array<uint32_t> *result,
30                        Context_ptr context,
31                        SlaveSession_ptr session,
32                        Event_ptr event) {
33      Uint32SharedArray_ptr arrContainer = event->first<
            Uint32SharedArray_ptr>(serialize::Uint32SharedArray);
34      *result = arrContainer->array();
35      olibClient->socketFacade()->service()->stop();
36  }
```

5.5.2 Shared communication code for any architecture

Listing 5.5 show the superclass for all our implementations for summing the two input arrays. Since we make this code manually, this is a natural abstraction to provide reuse of code, however a P2G compiler might solve this differently. The `SumArrays`-class contains code to deserialize input arrays, and code to send the result of summing the two arrays back to the client when the result is ready. The constructor takes the event containing the two input arrays as parameter. `SumArrays` stores the two input arrays internally as `arrayA` and `arrayB`, and it allocates a `result`-array of appropriate size for storing the result of summing the two input arrays.

Listing 5.5: SumArrays.h - Shared communication code for all sum arrays implementations

```
1  #pragma once
2  #include "olib/olib.h"
3
4  using namespace olib::protocol;
5  using namespace sevent;
6  using namespace sevent::event;
7  using sevent::datastruct::Uint32SharedArray;
8  using sevent::datastruct::Uint32SharedArray_ptr;
9
10 /** Shared code to parse the event flow for all
11  * sum arrays implementations. */
12 class SumArrays {
13     public:
14         SumArrays(Event_ptr event) {
15             Uint32SharedArray_ptr arrContainerA = event->at<
                   Uint32SharedArray_ptr>(0, serialize::
                   Uint32SharedArray);
16             Uint32SharedArray_ptr arrContainerB = event->at<
                   Uint32SharedArray_ptr>(1, serialize::
                   Uint32SharedArray);
17             arrayA = arrContainerA->array();
18             arrayB = arrContainerB->array();
19             arraySize = arrContainerA->size();
20             result = boost::shared_array<uint32_t>(
21                             new uint32_t[arraySize]);
22         }
23
24         void sumAndSendResult(Context_ptr context,
25                               SlaveSession_ptr session) {
26             sumTwoArrays();
27
28             Uint32SharedArray_ptr resultContainer;
29             resultContainer = boost::make_shared<Uint32SharedArray>(
                   result, arraySize);
30             Buffer_ptr resultBuf = Buffer::make(
31                     resultContainer,
32                     serialize::Uint32SharedArray);
33             Event_ptr resultEvent = Event::make(
34                     "example::ArraySumResult", resultBuf);
35             session->sendEvent(resultEvent);
36             context->unScheduleFunction();
```

```
37          }
38      protected:
39          /** Subclasses sum arrayA and arrayB, and
40           * store the results in result. */
41          virtual void sumTwoArrays() = 0;
42      protected:
43          boost::shared_array<uint32_t> result;
44          boost::shared_array<uint32_t> arrayA;
45          boost::shared_array<uint32_t> arrayB;
46          unsigned arraySize;
47  };
```

5.5.3 A standard C++ implementation for summing two arrays

Listing 5.6 shows the code for a shared library for normal CPUs programmable using C++. The `sumArrays`-function is the function that is loaded by the *run handler* as explained in Section 5.4.2. This function adds the `onArrayData` as handler for incoming arrays of data, and calls the `notifyInvoker` method as explained in Section 5.4.2. The `onArrayData`-method creates a object of the `CpuSumArrays` class, which is a subclass of `SumArrays`. The actual code for summing two arrays is in the `sumTwoArrays`-function.

Listing 5.6: Standard C++ implementation

```
1  #include <iostream>
2  #include <stdint.h>
3  #include "sevent/sevent.h"
4  #include "olib/olib.h"
5  #include "SumArrays.h"
6
7  class CpuSumArrays : public SumArrays {
8      public:
9          CpuSumArrays(sevent::event::Event_ptr event) :
10             SumArrays(event) {}
11     protected:
12         virtual void sumTwoArrays() {
13             for(int i = 0; i < arraySize; i++)
14             {
15                 result[i] = arrayA[i] + arrayB[i];
```

```
16              }
17          }
18 };
19
20 void onArrayData(olib::protocol::Context_ptr context,
21                  olib::protocol::SlaveSession_ptr session,
22                  Event_ptr event) {
23     CpuSumArrays sumArrays(event);
24     sumArrays.sumAndSendResult(context, session);
25 }
26
27 extern "C" {
28     void sumArrays(olib::protocol::Context_ptr context) {
29         std::cout << "sumArrays called" << std::endl;
30         context->addEventHandler("example::ArrayData",
31                                  onArrayData);
32         context->notifyInvoker();
33     }
34 }
```

5.5.4 CUDA and OpenCL implementations

CUDA and OpenCL implementations are available in the *examples/sumArrays/-sumArrays.olib* sub-directory in the *Olib* source-code (see Appendix B). The only significant changes in these implementations from the one in Listing 5.6 is the `sumTwoArrays`-method on line 12. In the OpenCL and CUDA implementations, the arrays are copied into the processing unit, and the code used to sum the two arrays are programmed using the respective kernel languages.

5.6 Automatic testing

Throughout this thesis we have had a fair amount of focus on automatic testing. One of the benefits of automatic tests becomes apparent when we consider the complexities of implementing many different implementations of the same code. Different implementations can result in different results. The differences may

be due to bugs. Bugs can easily be detected by automatic tests. A more subtle source of different results comes from the architectural differences between our target architectures. Processing units operate with varying precision. For example, many NVIDIA GPUs provide very fast floating point calculations [70] at the cost of precision.

Varying results from execution on each target architecture can affect the correctness of our results. Automatic tests can automate the task of verifying that each target architecture provides results within bounds acceptable for each P2G program.

5.7 Distributing P2G kernel instances using *Olib*

We assume we can compile P2G kernel definitions into shared libraries for each processing unit in our *target architecture*, and organize them into a directory-based application format as described in Chapter 3. Each P2G kernel instance can be executed by the P2G runtime as an application using the API presented in this chapter.

All kernel and scheduler communication is executed on the CPU. Some processing units, such as a GPU, requires data to be copied from the CPU to the processing unit before executing a P2G kernel. For very fast-running kernels, copying data may be a bottleneck. Therefore, we recommend that the P2G kernel language compiler and scheduler work together to minimize the need to communicate with the CPU. One possible strategy is to generate kernels that takes the number of sequential executions as parameter, thus allowing the processing unit to process multiple kernel instances without relinquishing control back to the CPU. Another possible solution is to extend *Olib* with the ability to communicate directly with the scheduler on some processing units, or at least to copy scheduling information into the processing units in the background while kernel instances execute.

In this chapter, we have presented an application as a client to a master. We have not made any research into the implementation of a fully fledged P2G runtime. However, we assume that the runtime will incorporate the same functionality

as a slave. Our slaves are standalone applications, but we do not know how the P2G runtime will be implemented or what additional requirements it may incorporate. Therefore, we designed *Olib* as a modular library that is flexible and relatively easy to change if required.

5.8 Summary

In this chapter we have demonstrated that adding support for distribution in *Olib* using *Sevent* is relatively easy. Furthermore, we have presented a viable solution for distribution of P2G kernel instances.

We have shown that making programs using *Olib* is labour intensive, which highlights the need for a higher level programming language, such as the P2G kernel language.

Integrating *Sevent* into *Olib* required very little work. Since *Olib* uses most of the features supported by *Sevent*, and these features are based on deduction of the requirements of the P2G project, we can conclude that *Sevent* is usable as the foundation for network communication in the P2G project.

Both *Sevent* and *Olib* work as intended from the programmers perspective, but without further proof of their viability for multimedia workloads we do not meet all goals defined in our Problem Definition. In the next chapter, we demonstrate that *Olib* works on both a single multi-core heterogeneous computer, and distributed in a network.

Chapter 6

Viability of multimedia workloads with *Olib*

In this section we demonstrate that *Olib* can be used for multimedia workloads on a single computer, and distributed to multiple computers in a network. In Chapter 5 we discussed a very simple example which calculates the sum of two arrays. Such an application is does not benefit much from GPU parallelization or network distribution, since data transfer is the most expensive operation. Therefore, we have selected a more computationally expensive workload as a proof of concept. This is a *proof of concept*, not a fully fledged benchmark of our implementation. A benchmark would require more workloads, and fine grained performance measurements of *Sevent* and *Olib*. The goal of this *proof of concept* is to demonstrate that our framework achieves performance-improvements when distributed in a network, and to highlight the complexities that the P2G scheduler must handle to achieve good performance.

6.1 Motion JPEG

Our workload is a Motion JPEG (MJPEG) encoder. MJPEG is a video format where each frame in the video is stored as a JPEG image. This workload is easy to parallelize, since each frame in the video can be processed independently.

A MJPEG encoder can be divided into 4 steps:

1. Read a raw video frame from disk into *three* arrays of data which makes up the image.

2. Perform Discrete cosine transform (DCT) [67] on each of the three arrays from step 1.

3. Rearrange the results from step 2 into the format required by the JPEG standard.

4. Append the JPEG image to the output video.

Step 2 is the most computationally expensive step, therefore we have chosen to optimize it as a *Olib* application. The other steps are computationally inexpensive compared to step 2, and they are performed in parallel with step 2. Therefore, steps 1, 2 and 4 does not affect the overall performance of encoding an entire video. The *Olib* application performs DCT on a single input array for each execution. The following implementations are provided:

CPU

This implementation only requires a standard C++ compiler. We compile this for Ubuntu Linux 64-bit X86, Apple Mac OSX 64-bit X86 and Ubuntu Linux 64-bit PowerPC. For X86 Ubuntu and Mac OSX, we have made two separate binaries. One is compiled with no special optimizations, making it usable on any 64-bit X86 compatible CPU. The other is optimized for modern Intel CPUs.

NVIDIA CUDA

This implementation is not very optimized, however it runs on any NVIDIA CUDA 1.1 compatible GPU.

OpenCL

This is a port of the CUDA implementation to OpenCL. It only works on NVIDIA GPUs.

The source code for these implementations are in the *examples/mjpeg/* directory in the *Olib* source code (see Appendix A).

6.2 How

We use custom made Python scripts to execute our tests. We time the complete execution time of each test program. The timer starts before spawning the process executing the test-program, and ends when the process completes. Our test-scripts execute each test *thirty times*, and we generate plots using shortest execution time. Execution time is measured as the time it takes to encode a single video frame. Storing the shortest time may be suboptimal, since average time with standard deviations may be more accurate in a production environment. However, we want to time our workloads under optimal conditions, and picking the best time after many executions should provide us with timing-results from very close to optimal conditions.

An overview of the test-computers is available in Appendix B. Our test-workload is executed on specific processing units on each computer, distributed over all available processing units on single computers, and distributed over a 1GB/s network to multiple test-computers.

6.3 MJPEG on a single computer

Figures 6.1, 6.2, 6.4, 6.3 and 6.5 shows the results of running the MJPEG encoder using different configurations on our test computers *Leela*, *Delano*, *Bush*, *Clinton* and *Cell-1*. In these tests, we configured our scheduler to use a single type of processing unit, such as CPU or GPU, at any given time. Since we have CPUs with multiple cores, we run all our CPU tests with 1, 2, 4, 6, 8, 10 and 12 instances in parallel. Each test is preceded by a test made without using *Olib*. These tests use the same code as the *Olib*-tests, but they do not use *Olib* for communication and parallelization.

In Figures 6.1, 6.2, 6.3 and 6.4, we can clearly see that binaries optimized for modern Intel CPUs, labeled *ModernIntel*, outperform binaries compiled for porta-

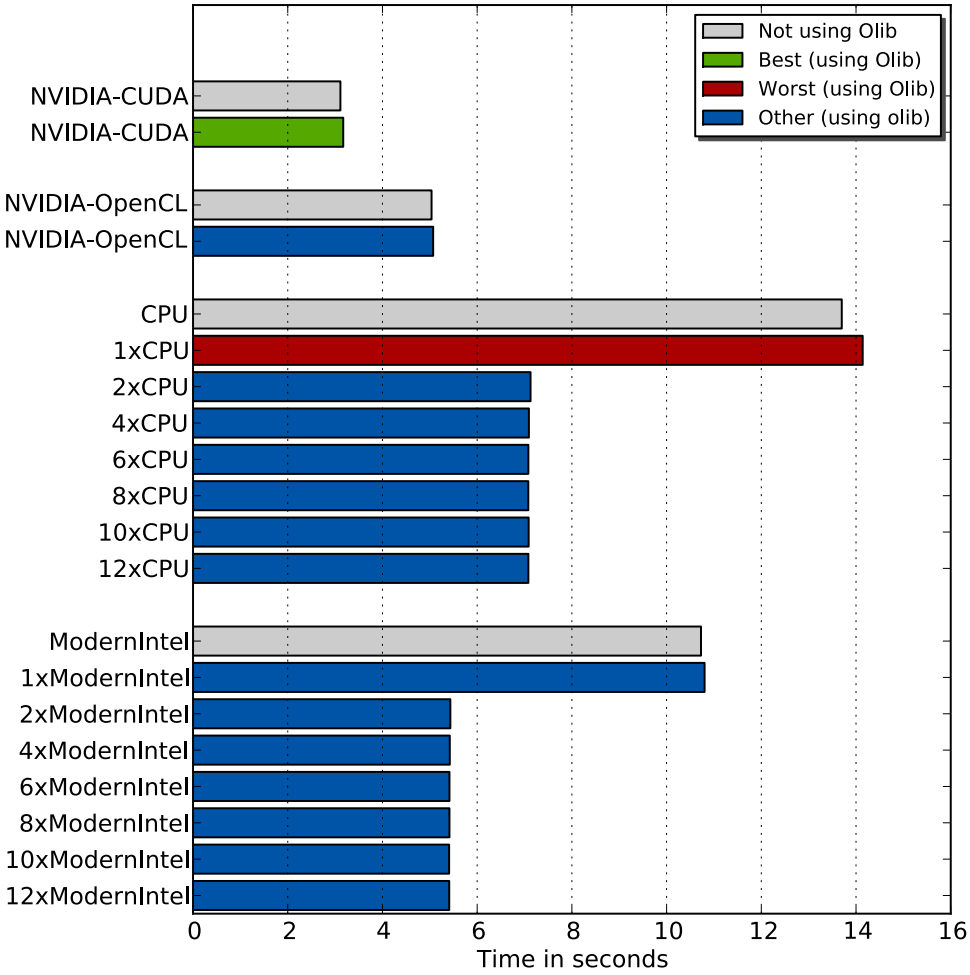

Figure 6.1: MJPEG with our implementations for each of the available processing units on *Delano*.

bility, labeled *CPU*. The tests labeled *ModernIntel* are compiled from exactly the same source code as the tests labeled CPU. The only difference between *CPU* and *ModernIntel* is the compile-flags used when compiling the source code into native binaries. This demonstrates the worth of optimized binaries, and the importance of providing a solution where it is practical to provide optimized binaries.

Figure 6.5 is primarily provided to show that *Olib* works on the PPE on a Playstation 3, which uses the CBE architecture. Our workload achieves very bad performance on the PPE, however PPE- support is the first step towards providing support for CBE-optimizations using *Olib*.

As long as the workload is parallelizable, the scheduler can distribute a work-

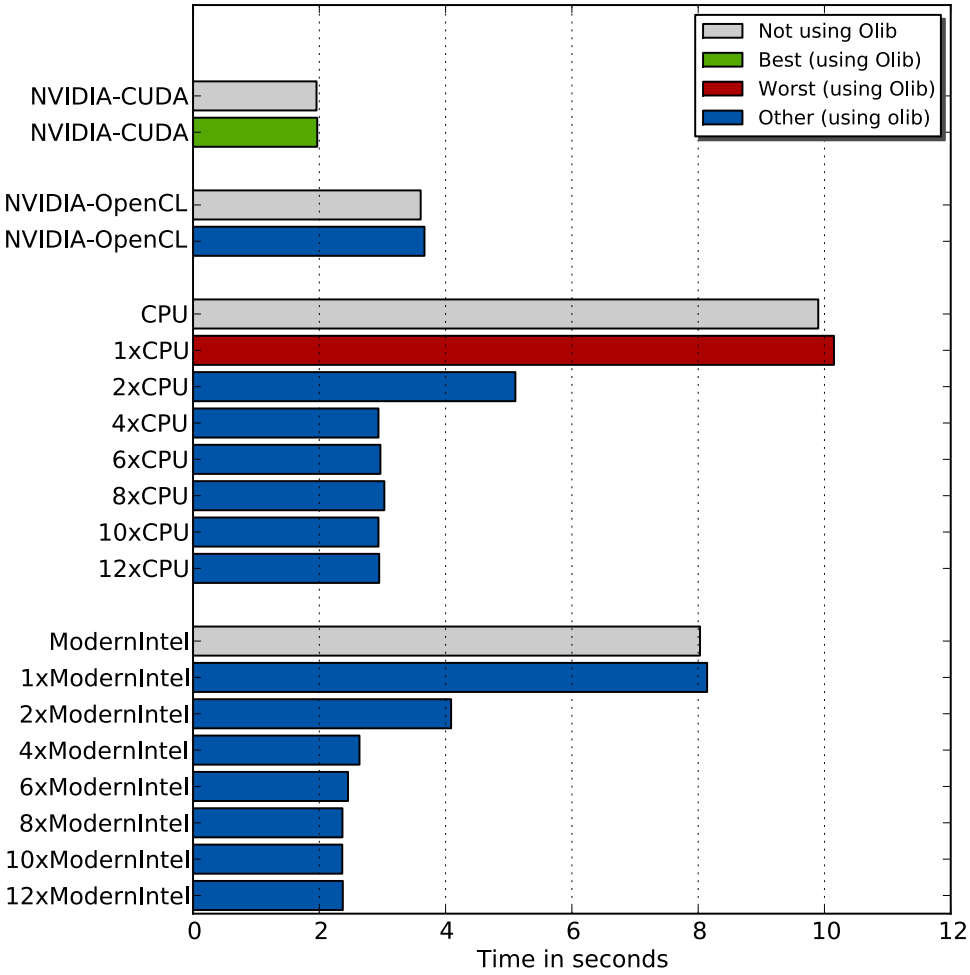

Figure 6.2: MJPEG with our implementations for each of the available processing units on *Bush*.

load over an appropriate number of processing units. We can clearly see that our workload achieves better performance when the scheduler distributes it over all the available processing cores in the CPUs. In Figure 6.4, we can also see why a dynamic scheduler is important. The CPU on *Leela* has only four cores, however it performs best, on our workload, with 12 instances in parallel. A static scheduler could easily have made the mistake of using only four instances in parallel on *Leela*. Since *Leela* is a laptop computer with power saving software and hardware, one may assume that the increased performance is caused by a gradual increase in available processing power as software or hardware detects a long running process intensive job. However, since we execute each test multiple times without any pauses that could allow power saving mechanisms to start,

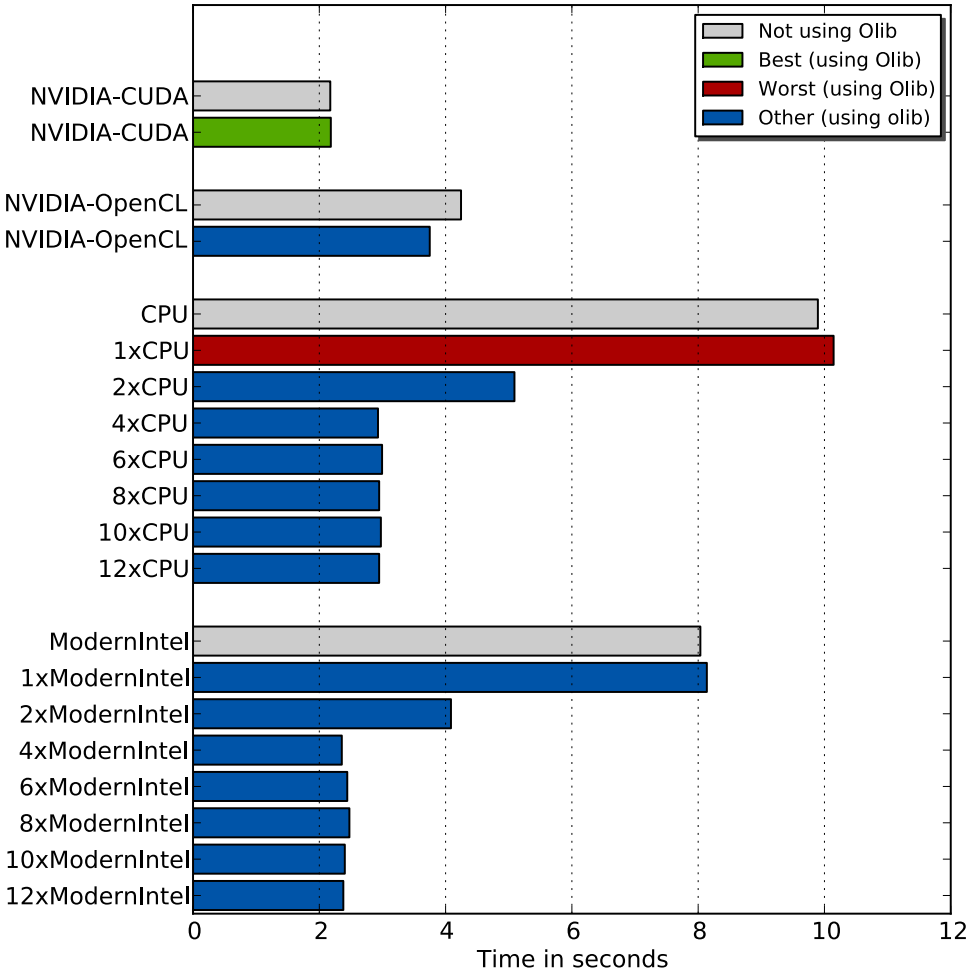

Figure 6.3: MJPEG with our implementations for each of the available processing units on *Clinton*.

this is highly unlikely. A much more probable reason for good multi-threading performance is that the Intel Core i7 processor is a modern processor optimized for multi-threading, using Intel Hyper-Threading Technology [106].

Figures 6.1, 6.2, 6.3, 6.4 and 6.5 show that *Olib* introduces a varying amount of overhead. The overhead varies, even with the same implementation, as we can see when comparing *NVIDIA-OpenCL* on *Delano* and *Bush*. In most cases the overhead is minimal, and we have not used profiling or other techniques to deduce the source of the overhead introduced by *Olib*. However we have theorized about several sources of overhead in the previous chapter, and we consider reduction of overhead a possible topic for future study.

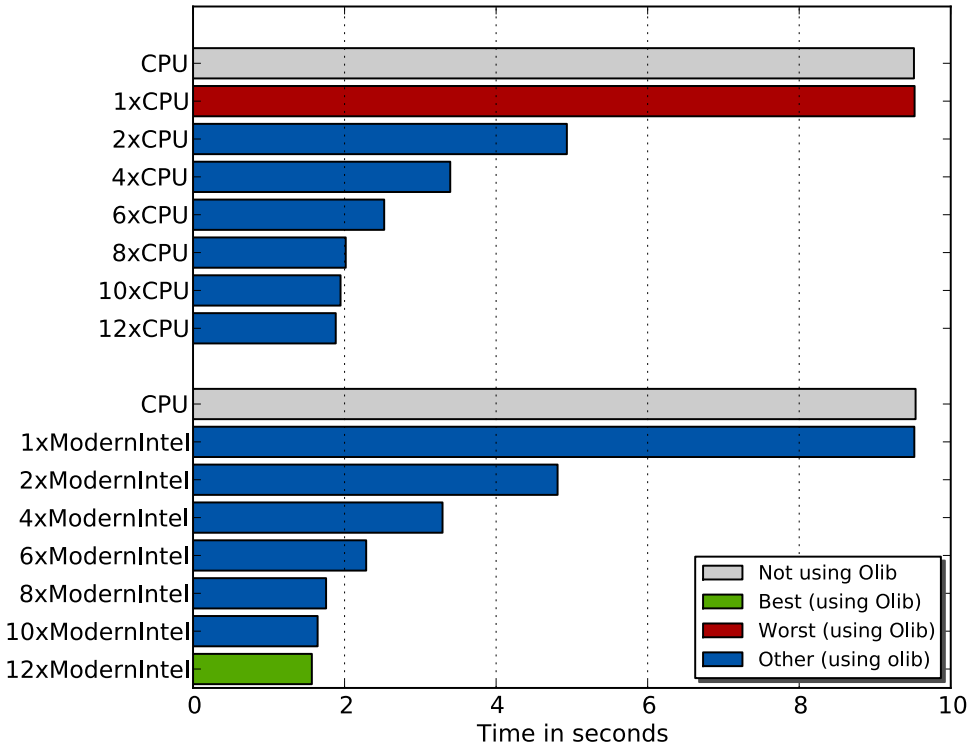

Figure 6.4: MJPEG with our implementations for each of the available processing units on *Leela*.

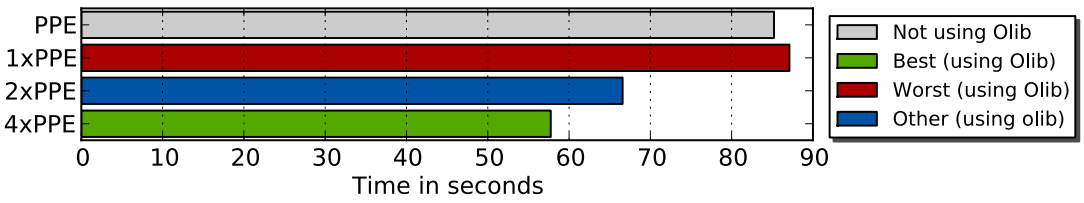

Figure 6.5: MJPEG with our implementations for each of the available processing units on *Cell-1*.

6.3.1 Using all available processing units

Figure 6.6 and Figure 6.7 shows the performance of our workload when we distribute it over all available processing units on *Delano* and *Bush*. The tests shows the performance achieved when using both their modern Intel CPU and their GPU at the same time. Both the OpenCL and CUDA implementations are shown. Neither the GPU on *Bush* nor the GPU on *Delano* support concurrent execution.

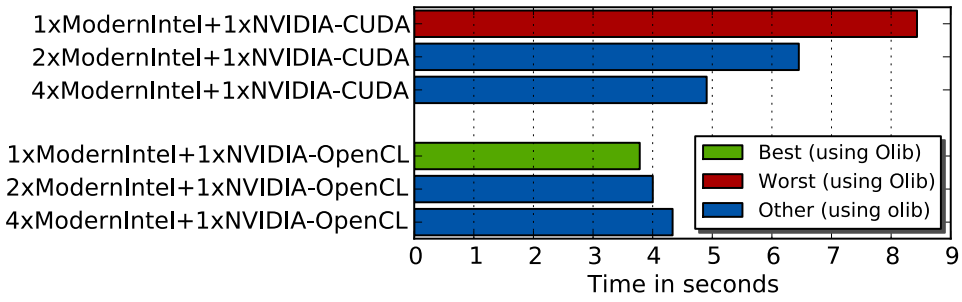

Figure 6.6: MJPEG distributed over all available processing units on *Delano*.

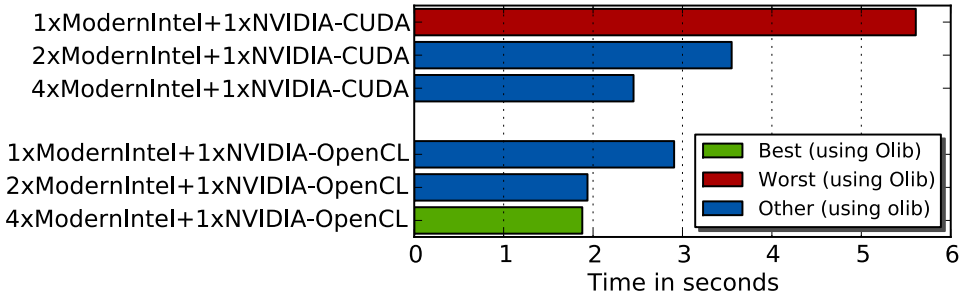

Figure 6.7: MJPEG distributed over all available processing units on *Bush*.

Therefore, we run our tests using a single GPU, and 1, 2 and 4 concurrent jobs on the CPU.

An interesting observation is that combining CUDA with the CPU leads to degraded performance compared to only using CUDA. Another interesting observation is that our OpenCL implementation is slower than our CUDA implementation, however when using CPU and OpenCL together on Bush, they achieve performance comparable to CUDA. This indicates that CUDA consumes much shared resources compared to OpenCL on NVIDIA GPUs. However, it may also indicate that our port from CUDA to OpenCL introduced more differences than we first assumed. Furthermore, these results show that performance can not be measured solely based on the performance measurements on each individual processing unit, since shared resources can become a bottleneck. This highlights the importance of a system, such as P2G, that aims to adapt to performance information dynamically at runtime.

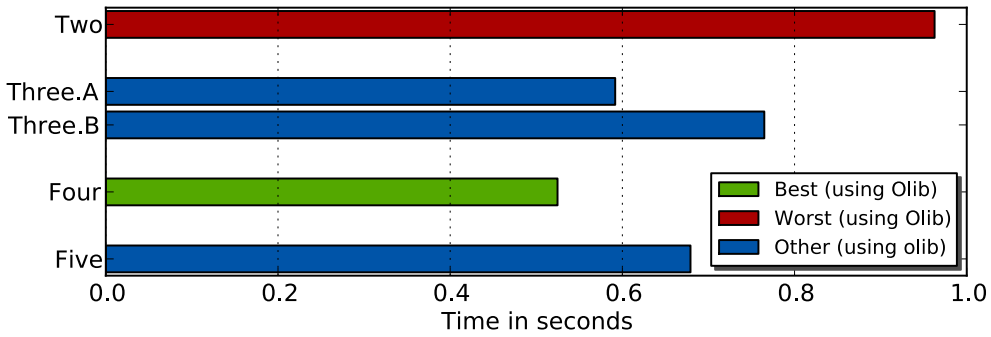

Figure 6.8: MJPEG distributed in a network.

Two	*Bush*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Clinton*:	4xModernIntel,1xNVIDIA-OpenCL
Three.A	*Bush*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Clinton*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Leela*:	12xModernIntel
Three.B	*Bush*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Clinton*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Delano*:	1xModernIntel,1xNVIDIA-OpenCL
Four	*Bush*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Clinton*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Delano*:	1xModernIntel,1xNVIDIA-OpenCL
	+ *Leela*:	12xModernIntel
Five	*Bush*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Clinton*:	4xModernIntel,1xNVIDIA-OpenCL
	+ *Delano*:	1xModernIntel,1xNVIDIA-OpenCL
	+ *Leela*:	12xModernIntel
	+ *Cell-1*:	PPE

6.4 Network-distributed MJPEG

Figure 6.8 shows the results of our tests when execution is distributed over more than one computer in a network. The figure is labeled with the number of computers used for each test, and the table below the figure shows the name of each of these computers and the processing units used.

We use the processing unit combinations that performs best when distributed over all available processing units on a single computer. The only exception is PPE on *Cell-1*, which is only included to demonstrate that it works. We only use one PPE, since the PPE is orders of magnitude slower than any of the other processing units among our test computers. Another interesting property of the PPE is that it is a *big endian* processor, while the other processing units are *little endian*. Notice that *Clinton* has the same processing units as *Bush*. Therefore, we use the same processing unit combinations for *Clinton* as for *Bush*.

When executing on a single computer, all four steps explained in Section 6.1 are performed on the same computer. In the distributed version, we read raw video frames on one computer (Step 1), distribute the DCT calculation (Step 2) over our test-computers, and rearrange results (Step 3) followed by append data to the output video (Step 4) on another computer. The computers used for Step 1 (the *data sender*), and Step3+4 (the *data receiver*) are two computers that is *not* any of our test-computers.

We can clearly see that network distribution of our workload improves performance. Our best result on a single computer is on *Bush* (Figure 6.7). In Figure 6.8, we can see that using both *Bush* and *Clinton* at the same time can almost double our performance. Adding a third and a fourth slave, continues to increases our performance, and we can clearly see that *Leela* (Three.A) performs better than *Delano* (Three.B). *Delano* is slower than *Leela* (compare Figure 6.6 to Figure 6.4), which shows that we scale with the available resources. Furthermore, Leela, Bush and Clinton use approximately 1.8 seconds encoding each video frame by themselves. When distributing our workload over all these computers (Figure 6.8, Three.A), we encode each video frame in 0.6 seconds. This is a linear performance increase.

Distributing over five computers decrease performance compared to four computers. This is because of the bad performance achieved by the PPE in *Cell-1* (see Figure 6.5). The only reason why this degrades performance to such a significant degree is because we have a static round-robin scheduler. The other computers perform as they do without *Cell-1*. The scheduler do not have any way of knowing that scheduling the PPE will be a very expensive operation. Therefore, it schedules DCT on the PPE even when it would be far more efficient for the overall result to just ignore it. This leads to degraded overall performance when

a single DCT calculation continues running on the PPE for a long time after DCT calculation on every other frame in the video has completed. Furthermore, the PPE is so slow compared to our other processing units, that it may not be possible to use it on some workloads, such as live video streaming and other workloads with deadlines. These issues highlight the need for future work on a dynamic scheduler with instrumentation data, and we believe our MJPEG workload on our test-computers would be a good combination for testing such a scheduler.

6.5 Summary

Our test-workload show that *Olib* can be used to distribute multimedia workloads. However, it is important to note that our MJPEG workload is only a proof of concept, and not a complete performance evaluation. Our results clearly show several potential benefits of *Olib*, especially as a part of P2G with a dynamic scheduler. These benefits include the ability to use the most efficient combination of all available processing units, the increased performance offered by native binaries optimized for specific CPUs, and potential for performance increases through network distribution. From our tests, we conclude that *Olib* is a viable solution for P2G.

Chapter 7

Conclusion

The goal of our research was to find methods enabling P2G applications to parallelize and distribute workloads, especially live multimedia workloads, in a network of many-core homogeneous and heterogeneous computers.

Sevent was created to be the foundation for all network communication in P2G. Integration of *Sevent* into P2G has started, and this process has shown that our analysis of the requirements of P2G was sound. We have found a viable solution (*Olib*) to one part of the larger challenge of integrating support for heterogeneous architectures into P2G. Furthermore, *Olib* combines an application format with support for heterogeneous architectures with *Sevent* to provide a viable solution for network distribution of P2G kernel instances. Our tests show that our solution has little overhead, and we have highlighted sources we believe lead to this overhead.

Furthermore, the *Sevent* library is a general purpose communication library that may be used in other projects. It is modular enough to allow for performance improvements and inclusion of additional features when required. The design is minimalistic, clean and well organized, which makes it easy to maintain as a stable foundation in any project, including P2G.

7.1 Future work

The most obvious continuation of our work is to integrate it into P2G. Furthermore, the following topics are worthy of further studies:

- Direct communication between kernel instances. At this point, the API forces processing units to relinquish control back to the CPU for communication with the scheduler. If this communication overhead could be removed, it would be possible to distribute jobs with very small granularity.
- Performance tests using different TCP implementations to assert if TCP-communication performance can be improved. This research may also include development of custom application layer protocols on top of UDP.
- Solutions to the issues explained in Section 5.3.
- Research into a separate process based solution instead of a shared library solution. This should require minimal changes to *Olib*. Only the modules responsible for loading and abstracting communication should be affected.
- Support for more platforms. Both *Olib* and *Sevent* have comprehensive automatic test suites. Both libraries have been compiled and tested on *Mac OS X 10.6*, *Ubuntu Linux 64-bit x86* and *Ubuntu Linux 64-bit PowerPC*. Both libraries should be portable to other platforms, however certain compatibility issues must be expected. The most obvious issue for portability is loading of shared libraries, which we explained in Chapter 3.
- Tests using other OpenCL-implementations than *NVIDIA OpenCL*.
- Minimize the performance overhead introduced by *Olib*.

Appendix A

Source code

The source code for *Sevent* is available from `http://github.com/espenak/sevent`. *Sevent* is released under the BSD license provided in LICENSE.txt at the same URL.

The *Olib* source code can be downloaded from `http://heim.ifi.uio.no/espeak/olib.tgz`.

Appendix B

Test-computers

We run our tests on a set of test-computers. In this Appendix, the most significant properties of these computers are listed.

B.1 Delano

CPU:	Intel(R) Core(TM)2 Duo @ 2.66GHz
Number of CPUs:	1
Number of CPU cores:	2
GPU:	Nvidia GeForce GT 220
Operating system:	Ubuntu 10.04 LTS 64-bit X86 (Linux 2.6.32-30-generic)

B.2 Bush and Clinton

CPU:	Intel(R) Core(TM) i5 @ 2.67GHz
Number of CPUs:	1
Number of CPU cores:	4
GPU:	Nvidia GeForce GTX 280
Operating system:	Ubuntu 10.04 LTS 64-bit X86 (Linux 2.6.32-24-generic)

B.3 Cell-1

Model name:	Sony Playstation 3
CPU:	Cell Broadband Engine, altivec supported, 6 usable SPEs and one PPE
Operating system:	Ubuntu 10.04 LTS 64-bit PowerPC (Linux 2.6.28-6-powerpc64-smp)

B.4 Leela

Model name:	Apple MacBook Pro 8,2
CPU:	Intel(R) Core(TM) i7 @ 2GHz
Number of CPUs:	1
Number of CPU cores:	4
Operating system:	Mac OS X 10.6.7 (Darwin 10.7.0)

References

[1] E. M Gordon. Cramming more components onto integrated circuits. *Electronics Magazine*, 4, 1965.

[2] T. Chen, R. Raghavan, J. N Dale, and E. Iwata. Cell broadband engine architecture and its first implementation—a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[3] S. Ryoo, C. I Rodrigues, S. S Baghsorkhi, S. S Stone, D. B Kirk, and W. W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, page 73–82, 2008.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

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

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, page 59–72, 2007.

[7] P. H Vrba, C. Griwodz, P. Beskow, and D. Johansen. The nornir run-time system for parallel programs using kahn process networks. In *2009 Sixth IFIP International Conference on Network and Parallel Computing*, page 1–8, 2009.

[8] M. I Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th*

international conference on Architectural support for programming languages and operating systems, page 151–162, 2006.

[9] P. B Beskow, H. Espeland, H. K Stensland, P. N Olsen, S. Kristoffersen, E. A Kristiansen, C. Griwodz, and P. Halvorsen. Distributed Real-Time processing of multimedia data with the P2G framework. *Eurosys*, (Poster Session), 2011.

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

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

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

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

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

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

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

[17] Michael I. Gordon, William Thies, and Saman 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.

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

[19] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

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

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

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

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, page 164–177, 2003.

[24] I. Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.

[25] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, page 41–46, 2005.

[26] W. Huang, J. Liu, B. Abali, and D. K Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing*, page 125–134, 2006.

[27] P. Apparao, S. Makineni, and D. Newell. Characterization of network processing overheads in xen. 2006.

[28] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the xen virtual machine monitor. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 24–24, 2005.

[29] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N Matthews. Xen and the art of repeated research. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 47–47, 2004.

[30] VMware. A performance comparison of hypervisors. `http://www.vmware.com/resources/techresources/711`, April 2011.

[31] J. N Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, page 6–es, 2007.

[32] L. Shi, H. Chen, and J. Sun. vCUDA: GPU accelerated high performance computing in virtual machines. 2009.

[33] E. Meijer and J. Gough. Technical overview of the common language runtime. *language*, 29(7), 2002.

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

[35] B. Blount and S. Chatterjee. An evaluation of java for numerical computing. *Computing in Object-Oriented Parallel Environments*, page 501–502, 1998.

[36] G. P. Nikishkov, Y. G Nikishkov, and V. V. Savchenko. Comparison of c and java performance in finite element computations. *Computers & structures*, 81(24-25):2401–2408, 2003.

[37] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *Micro, IEEE*, 17(3):36–43, 1997.

[38] J. E Moreira, S. P Midkiff, and M. Gupta. A comparison of java, C/C++, and fortran for numerical computing. *Antennas and Propagation Magazine, IEEE*, 40(5):102–105, 1998.

[39] S. Marr, M. Haupt, S. Timbermont, B. Adams, T. D'Hondt, P. Costanza, and W. De Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. *Arxiv preprint arXiv:1002.0939*, 2010.

[40] A. Noll, A. Gal, and M. Franz. CellVM: a homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In *Workshop on Cell Systems and Applications*, 2008.

[41] K. Williams, A. Noll, A. Gal, and D. Gregg. Optimization strategies for a java virtual machine interpreter on the cell broadband engine. In *Proceedings of the 5th conference on Computing frontiers*, page 189–198, 2008.

[42] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: a programmer-friendly interface for accelerating java programs with CUDA. *Euro-Par 2009 Parallel Processing*, page 887–899, 2009.

[43] E. Youngdale. Kernel korner: The ELF object file format by dissection. *Linux Journal*, 1995(13es):15, 1995.

[44] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification, May 1995.

[45] M. Pietrek. Peering inside the PE: a tour of the win32 (R) portable executable file format. *Microsoft Systems Journal-US Edition*, page 15–38, 1994.

[46] B. W Kernighan, D. M Ritchie, and P. Ejeklint. *The C programming language*, volume 78. Citeseer, 1988.

[47] Z. Yang and K. Duddy. CORBA: a platform for distributed object computing. *SIGOPS Operating Systems Review*, 30(2):4–31, 1996.

[48] D. W Walker and J. J Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[49] R. Love. Get on the D-BUS. *Linux Journal*, 2005(130):3, 2005.

[50] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F Nielsen, S. Thatte, and D. Winer. *Simple object access protocol (SOAP) 1.1*. May, 2000.

[51] B. Stroustrup. An overview of c++. In *ACM Sigplan Notices*, volume 21, page 7–18, 1986.

[52] J. Norton. Dynamic class loading in c++. *Linux Journal*, 2000(73es):38, 2000.

[53] M. A Ellis and B. Stroustrup. *The annotated C++ reference manual*. Pearson Education India, 1994.

[54] Y. T Li, D. Leith, and R. N Shorten. Experimental evaluation of TCP protocols for high-speed networks. *IEEE/ACM Transactions on Networking (ToN)*, 15(5):1109–1122, 2007.

[55] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[56] A. Petlund, K. Evensen, C. Griwodz, and P. Halvorsen. TCP mechanisms for improving the user experience for time-dependent thin-stream applications.

In *Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on*, page 176–183, 2008.

[57] M. Henning. The rise and fall of CORBA. *Queue*, 4(5):28–34, 2006.

[58] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, October 1994.

[59] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, page 287–301, 2001.

[60] E. Collar and R. Valerdi. Role of software readability on software development cost. In *Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, Herndon, VA*, October 2006.

[61] E. H\ost and B. \Ostvold. Debugging method names. *ECOOP 2009–Object-Oriented Programming*, page 294–317, 2009.

[62] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[63] R. C Martin. Clean code: a handbook of agile software craftsmanship. *Prentice Hall PTR Upper Saddle River, NJ, USA*, page 448, 2008.

[64] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, February 2001.

[65] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 513-516, 2 edition, May 1998.

[66] Nvidia. Nvidia CUDA programming guide 3.2, August 2010.

[67] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transfom. *Computers, IEEE Transactions on*, 100(1):90–93, 1974.

References from the Internet

[68] Moore's law: Made real by intel innovations. http://www.intel.com/technology/mooreslaw/index.htm, April 2011.

[69] IBM. Cell broadband engine architecture, 2006.

[70] Nvidia. Nvidia CUDA programming guide 3.2, August 2010.

[71] Nvidia. CUDA zone. http://www.nvidia.com/object/cuda_home_new.html, May 2011.

[72] Khronos. OpenCL. http://www.khronos.org/opencl/, May 2011.

[73] Intel® OpenCL SDK - intel® software network. http://software.intel.com/en-us/articles/intel-opencl-sdk/, April 2011.

[74] alphaWorks : OpenCL development kit for linux on power : Overview. http://www.alphaworks.ibm.com/tech/opencl, May 2011.

[75] VMware. VMware workstation. http://www.vmware.com/products/workstation/index.html, May 2011.

[76] List of JVM languages - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/List_of_JVM_languages, April 2011.

[77] List of CLI languages - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/List_of_CLI_languages, April 2011.

[78] CUDA.NET. http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx, April 2011.

[79] OpenCL.NET. http://www.hoopoe-cloud.com/Solutions/OpenCL.NET/Default.aspx, April 2011.

[80] The open toolkit library | OpenTK. http://www.opentk.com/, April 2011.

[81] Java bindings for the OpenCL API. http://jogamp.org/jocl/www/, April 2011.

[82] Fat binary - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Fat_binary, April 2011.

[83] Universal binary - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Universal_Binaries, April 2011.

[84] Mach-O - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mach-O, May 2011.

[85] FatELF. http://icculus.org/fatelf/, April 2011.

[86] FatELF: i've been so damned tired. http://icculus.org/cgi-bin/finger/finger.pl?user=icculus&date=2009-11-03&time=19-08-04, April 2011.

[87] Oracle. JAR file specification. http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html, May 2011.

[88] Apple. Code loading programming topics: About loadable bundles. http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/LoadingCode/Concepts/AboutLoadableBundles.html, May 2011.

[89] LoadLibrary function (Windows). http://msdn.microsoft.com/en-us/library/ms684175(v=vs.85).aspx, April 2011.

[90] Apple. Mac OS x manual page for dlclose(3). http://developer.apple.com/library/ios/#documentation/system/conceptual/manpages_iphoneos/man3/dlclose.3.html, May 2011.

[91] Boost.Serialization. http://www.boost.org/doc/libs/1_46_1/libs/serialization/doc/index.html, March 2011.

[92] Protocol buffers - google's data interchange format. http://code.google.com/p/protobuf/, March 2011.

[93] Apache thrift. http://incubator.apache.org/thrift/, March 2011.

[94] RFC 1832 - XDR: external data representation standard. http://tools.ietf.org/html/rfc1832, March 2011.

[95] epoll(7) - linux manual page. http://www.kernel.org/doc/man-pages/online/pages/man4/epoll.4.html, May 2011.

[96] libevent. http://monkey.org/~provos/libevent/, April 2011.

[97] Boost.Asio - boost 1.46.0. http://www.boost.org/doc/libs/1_46_0/doc/html/boost_asio.html, April 2011.

[98] Bjarne Stroustrup. C++0x FAQ. http://www2.research.att.com/~bs/C++0xFAQ.html, May 2011.

[99] Boost.Thread - boost 1.46.1. http://www.boost.org/doc/libs/1_46_1/doc/html/thread.html, April 2011.

[100] Devilry project website. http://devilry.github.com/, March 2011.

[101] Kdelibs coding style. http://techbase.kde.org/Policies/Kdelibs_Coding_Style#Variable_declaration, March 2011.

[102] Google c++ style guide. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#General_Naming_Rules, March 2011.

[103] shared_ptr - boost 1.46.1. http://www.boost.org/doc/libs/1_46_1/libs/smart_ptr/shared_ptr.htm, April 2011.

[104] Boost.Any - boost 1.46.1. http://www.boost.org/doc/libs/1_46_1/doc/html/any.html, April 2011.

[105] Boost.Test - boost 1.46.1. http://www.boost.org/doc/libs/1_46_1/libs/test/doc/html/index.html, April 2011.

[106] Intel. Intel core processor family. http://www.intel.com/consumer/products/processors/core-family.htm, May 2011.