A PORTABLE RUNTIME INTERFACE FOR MULTI-LEVEL MEMORY HIERARCHIES

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Michael C. Houston

March 2008

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick M. Hanrahan
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William J. Dally

Approved for the University Committee on Graduate Studies.

Abstract

The efficient use of a machine's memory system and parallel processing resources has become one of the most important challenges in program optimization. Moreover, efficient use of the memory hierarchy is increasingly important because of the power cost of data transfers through the system. Architecture trends are leading to large scale parallelism using simpler cores and progressively deeper and complex memory hierarchies. These new architecture designs have improved power characteristics and can offer large increases in performance, but traditional programming techniques are inadequate for these architectures.

In this dissertation, we explore a programming language and runtime system for making efficient use of the memory hierarchy and parallel processing resources. This dissertation provides an overview of Sequoia, a programming language we have developed at Stanford to facilitate the development of memory hierarchy aware parallel programs that remain portable across modern machines featuring different memory hierarchy configurations. Sequoia abstractly exposes hierarchical memory in the programming model and provides language mechanisms to describe communication vertically through the machine and to localize computation to particular memory locations within it.

This dissertation presents a platform independent runtime interface for moving data and computation through parallel machines with multi-level memory hierarchies. We show that this interface can be used as a compiler target for the Sequoia language and compiler, and can be implemented easily and efficiently on a variety of platforms. The interface design allows us to compose multiple runtimes, achieving portability across machines with multiple memory levels. We demonstrate portability of Sequoia programs across machines with two memory levels with runtime implementations for multi-core/SMP machines, the STI Cell Broadband Engine, a distributed memory cluster, and disk systems. We also demonstrate portability across machines with multiple memory levels by composing runtimes and running on a cluster of SMP nodes, out-of-core algorithms on a Sony Playstation 3 pulling data from disk, and a cluster of Sony Playstation 3's. All of this is done without any source level modifications to the Sequoia program. With this uniform interface, we achieve good performance for our applications and maximize bandwidth and computational resources on these system configurations.

Acknowledgments

I first need to thank all of my research collaborators, especially those on working with me on the Sequoia project: Ji-Young Park, Manman Ren, Tim Knight, Kayvon Fatahalian, Mattan Erez, and Daniel Horn. I largely attribute the success of the project to all the hard work and long nights required to get all the code written and papers out. Without Kayvon, who lead the language development, and Tim Knight, who lead the compiler work, we never would have been able to get to level where the runtime work was viable. For the last year, Ji-Young and Manman have really been the folks doing a lot of heavy lifting to get all the compiler infrastructure converted over to the runtime system and help get all the applications optimized and tested.

I'd also like to thank Pat Hanrahan for taking a risk and funding me as a Masters student, and then supporting me for entrance into the Ph.D program. Working with Pat has taught me a great deal about doing research and standing up for my ideas. Although my research interests in the end diverged from Pat's area of interest, he was supportive of my pursuits and helped me engage with other groups.

Alex Aiken has gone above and beyond for me and the rest of the people working in the Sequoia project. Although we were not his students, he would meet with us every week, if not more, to help us codify the research goals and directions for the Sequoia project and force us to really work through the issues and to set and make deadlines. Alex was also deeply involved in all of the Sequoia publications, even editing papers and my dissertation while on his "year off" traveling around the world.

I have had many influences while at Stanford which contributed to my success. My first year was spent under the wing of Greg Humphreys, while although an interesting character and often a distraction, really taught me how to handle working on large systems projects and survive with humor. Working with Ian Buck on the Brook project and then subsequent work on GPGPU gave me perhaps more than my fair share of visibility, which in turn helped to secure later funding, fellowships, and solid job offers. Ian was of great help reassuring me that I really would finish and get everyone to read and sign my dissertation. Jeremy Sugerman often served as the voice of pragmatism, and sometimes sanity; he approaches most things with sarcasm and humor making sure things arent put up on a pedestal, and is quick to call people on being vague or fluffy when a deeper explanation is really required. Daniel Horn is an amazing hacker who was the first real user of the Sequoia system and worked with me on many papers and system projects. Kayvon Fatahalian is an amazing researcher who has made me really think carefully through my research and served as my primary sounding board for all my various research ideas and proposed solutions.

I have had a variety of funding sources throughout my career including DOE, ASC, LLNL, ATI, IBM, and Intel. Specifically, I'd like to thank Randy Frank, Sean Ahern, and Sheila Vaidya from LLNL and Allen McPherson and Pat McCormick from LANL for supporting much of my research as well as interacting closely with us on projects. I'd also like to thank Bob Drebin, Eric Demers, and Raja Koduri from ATI/AMD for letting me come in to ATI for internships and consulting and be disruptive, in a good way. I was able to learn a tremendous about GPU architectures, memory systems, working with large teams, and how to hold my ground when I think I'm on the right path. IBM was gracious in providing us with early access to IBM Cell blade systems to start the Sequoia project, a large reason for the early success of the project. I was honored to receive a graduate fellowship from Intel for my final two years of research. This fellowship allowed me to concentrate purely on the Sequoia project as well as interact with some amazing people at Intel.

I would like to thank my committee members, Vijay Pande and Mendel Rosenblum, for providing feedback on my research and willingness to sit on my committee. Special thanks to my readers, Pat Hanrahan, Alex Aiken, and Bill Dally for helping me get everything into a coherent text and present the reams of numbers in a reasonable way. Bill got back to me on my dissertation much faster than expected and really helped to clean up some of the reasoning.

Being at Stanford has been an amazing experience, and I'm glad I took the risk to come here. The quality of the students and faculty is just simply amazing, and I was constantly challenged as well as humbled. It's amazing when someone outside of your area can quickly understand your research, and poke holes in your assumptions, and help you to better mold how you present and talk about things. Being in the graphics lab is a valuable, if not intimidating, experience. I think our lab is incredibly hard on speakers, but the feedback is very constructive and helps you to develop a thick skin, a requirement for both academia and industry, and quickly improve how you present your research. I was extremely hard on infrastructure and resources for my research, so I'd especially like to thank Ada Glucksman, Heather Gentner, John Gerth, Joe Little, and Charlie Orgish for being there when I needed them for help.

Last, but not least, I'd like to thank my family. My parents have been very supportive through my education, even during the dark times when I as really struggling, and have been willing to put themselves on the line for me when it has come to my education. Id like to thank my sister Janice for saving my life more than once and putting up with me when I used to really pester her. I'd also really like to thank my wife's family, James, Ruth and

Aaron for welcoming me into their home during my time at Stanford. I'd like to especially thank my mother-in-law for preparing so many meals for me when I have been busy. And most importantly, I need to thank my loving wife, Tina, who has been there for me during the best and worst of times. She understood during paper crunches when I'd disappear for weeks and when I was not in the best of moods. She was even patient when forced to read multiple drafts of my papers and this dissertation. I could not have gotten through everything without her support and love.

Thank you all!

For all those who helped me along the way, this would not have been possible without you...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current programming languages and runtime systems do not provide the mechanisms necessary to efficiently manage data movement through the memory hierarchy or efficiently manage the parallel computational resources available in the machine. Moreover, previous research has a limited degree of portability across different architectures because of built-in assumptions about the underlying hardware capabilities. Sequoia has been designed to allow efficient use of the memory system and parallel computational resources while providing portability across different machine types and efficient control of complex memory hierarchies.

Most parallel programs today are written using a *two-level* memory model, in which the machine architecture, regardless of how it is physically constructed, is abstracted as a set of sequential processors executing in parallel. Consistent with many parallel programming languages, we refer to the two memory levels as *local* (local to a particular processor) and *global* (the aggregate of all local memories). Communication between the global and local levels is handled either by explicit message passing (as with MPI [MPIF, 1994]) or by language-level distinctions between local and global references (as in UPC [Carlson et al., 1999] and Titanium [Yelick et al., 1998]). Using a two-level abstraction to program a *multi-level* system, a configuration with more than one level of communication, obscures details of the machine that may be critical to performance. On the other hand, adding support outside of the programming model for moving computation and data between additional levels leads to a multiplicity of mechanisms for essentially the same functionality (e.g., the ad hoc or missing support for out-of-core programming in most two-level systems). It is our thesis that programming abstractions, compilers, and runtimes directly supporting multi-level machines are needed.

This work is based on the belief that three trends in machine architecture will continue for the foreseeable future. First, future machines will continue to increase the depth of the memory hierarchy, making direct programming model support for more than two-level systems important. Second, partly as a result of the increasing number of memory levels, the variety of communication protocols for moving data between memory levels will also continue to increase, making a uniform communication API desirable both to manage the complexity and improve the portability of applications. Lastly, architectures requiring explicit application control over the memory system, often through explicit memory transfers, will become more common. A current extreme example of this kind of machine is LANL's proposed Roadrunner machine, which combines disk, cluster, SMP, and the explicit memory control required by the Cell processor [LANL, 2008].

In this thesis, we present an API and runtime system that virtualizes memory systems, giving a program the same interface to data and computation whether the memory level is a distributed memory, a shared memory multiprocessor (SMP), a single processor with local memory, or disk, among other possibilities. Furthermore, this API is composable, meaning that a runtime for a new multi-level machine can be easily constructed by composing the runtimes for each of its individual levels.

The primary benefit of this approach is a substantial improvement in portability and ease of maintenance of a high performance application for multiple platforms. Consider, for example, a hypothetical application that is first implemented on a distributed memory cluster. Typically, such a program relies on MPI for data transfer and control of execution. Tuning the same application for an SMP either requires redesign or reliance on a good shared memory MPI implementation. Unfortunately, in most cases the data transfers required on the cluster for correctness are not required on a shared memory system and may limit achievable performance. Moving the application to a cluster of SMPs could use a MPI process per processor, which relies on a MPI implementation with recognition of which processes are running on the same node and which are on other nodes to orchestrate efficient communication. Another option is to use MPI between nodes and Pthreads or OpenMP compiled code within a node, thus mixing programming models and mechanisms for communication and execution. Another separate challenge is supporting out-of-core applications which need to access data from disk, which adds yet another interface and set of mechanisms that need to be managed by the programmer. As a further complication, processors that require explicit memory management, such as the STI Cell Broadband Engine, present yet another interface that is not easily abstracted with traditional programming techniques.

Dealing with mixed mode parallel programming and the multiplicity of mechanisms and abstractions makes programming for multi-level machines a daunting task. Moreover, as bandwidth varies through the machine, orchestrating data movement and overlapping communication and computation become difficult.

The parallel memory hierarchy (PMH) programming model provides an abstraction of multiple memory levels [Alpern et al., 1993]. The PMH model abstracts parallel machines as trees of memories with slower memories toward the top near the root, faster memories toward the bottom, and with CPUs at the leaves. The Sequoia project has created a full language, compiler, runtime system, and a set of applications based on the PMH model [Fatahalian et al., 2006; Knight et al., 2007; Houston et al., 2008]. The basic programming construct in Sequoia is a *task*, which is a function call that executes entirely in one level of the memory hierarchy, except for any *subtasks* that task invokes. Subtasks may execute in lower memory levels of the system and recursively invoke additional subtasks at even lower levels. All task arguments, including arrays, are passed by value-result (i.e., copy-in, copy-out semantics). Thus, a call from a task to a subtask represents bulk communication, and all communication in Sequoia is expressed via task calls to lower levels of the machine. The programmer decomposes a problem into a tree of tasks, which are subsequently mapped onto a particular machine by a compiler using a separate *mapping* dictating which tasks are to be run at which particular machine levels.

Although our early Sequoia work demonstrated applications running on IBM Cell blades and a cluster of PCs, it did not show portability to multi-level memory hierarchies. More importantly this earlier work also relied on a custom compiler back-end for Cell and a complex and advanced runtime for a cluster of PCs which managed all execution and data movement in the machine through a JIT mechanism. The difficulty with this approach is that every new architecture requires a monolithic, custom backend and/or a complex runtime system.

The Sequoia compiler, along with the bulk optimizations and custom backend used for Cell, is described by Knight et al. [Knight et al., 2007]; the Sequoia language, programming model, and cluster runtime system is described by Fatahalian et al. [Fatahalian et al., 2006]. In this dissertation, we build on the previous PMH and Sequoia work, but we take the approach of defining an abstract runtime interface as the target for the Sequoia compiler and provide separate runtime implementations for each distinct kind of memory in a system. As discussed above, our approach is to define a single interface that all memory levels support.

Since these interfaces are composable, adding support for a new architecture only requires assembling an individual runtime for each adjacent memory level pair of the architecture rather than reimplementing the entire compiler backend.

1.1 Thesis Contributions

This dissertation explores the design and development of an abstract machine model and runtime system for efficiently programming parallel machines with multi-level memory hierarchies. We make several contributions in the areas of computer systems, parallel programming, machine abstractions, and portable runtime systems outlined below. Our approach is to define a single interface that provides one abstraction for communication and control between multiple levels in a memory hierarchy. Since these interfaces are composable, adding support for a new architecture only requires assembling an individual runtime for each adjacent memory level pair of the architecture rather than reimplementing a specialized program for each machine or a custom compiler backend.

Abstract machine model for parallel machines We present a uniform scheme for explicitly describing memory hierarchies. This abstraction captures common traits important for performance on memory hierarchies. We formalize previous theoretical models and show how the proposed abstraction can be composed to allow for the execution on machines with multiple levels of memory hierarchy.

Portable runtime API We discuss the development and implementation of a runtime API that can be mapped to many system configurations. This interface allows a compiler to optimize and generate code for a variety of machines without knowledge of the specific bulk communication and execution mechanisms required by the machine configuration. We explore the abstraction by evaluating the efficiency of implementations on several common parallel system configurations.

1.2 Outline

The centerpieces of this thesis are the abstract machine model and runtime interface for memory hierarchies, enabling Sequoia to run on multiple architectures, its implementation on various architectures, and the analysis of the portability and efficiency of the abstraction on multiple platforms, including the cost of mapping the abstraction to each platform. The Sequoia language and complete system are discussed, but the focus of the discussion is on the features of the language and the design decisions made along the way to preserve portability and maintain performance on our platforms. Some of these decisions directly impact the types of applications that can be written easily in Sequoia and executed efficiently on our runtime systems. These unintended consequences are discussed in the discussion (Chapter 7).

Chapter 2

Background

2.1 Architecture Trends

2.1.1 Memory Systems

In modern architectures, the throughput and latency of the main memory system is much lower than the rate at which the CPU can execute instructions. This limits the effective processing speed when the processor is required to perform minimal processing on large amounts of data, the processor must continuously wait for data to be transferred to or from memory. As the difference between compute performance and memory performance continues to widen, many algorithms quickly become bound by memory performance rather than compute performance. This effect is known as the von Neumann bottleneck. When many abstract models of computation were created, compute performance was the bottleneck. As VLSI scaling and processor technologies have improved, we can perform computation at much faster rates than we can read from main memory. For example, the Intel

Core2Duo Quad (QX9650) can perform computation at 96 GFLOPS and yet has $\sim$5 GB/s of bandwidth to main memory. We have already passed an order of magnitude difference between our compute capability and bandwidth to main memory and the gap is continuing to widen.

Caches have reduced the effects of the von Neumann bottleneck, but in an effort to keep the computational units of the processor busy, processors have gained multiple levels of cache, thus building a memory hierarchy. Processors now have multiple levels of caches with high-bandwidth, low latency, but small caches close to the processor, and lower-bandwidth, higher latency, larger caches further away. For optimum performance, even on simple applications, the user must make efficient use of all the caches in the hierarchy. This is generally done by carefully blocking data into the caches to maximize the amount of reuse. For example, the first-order optimization effect for matrix multiply over the naive triple nested for loop implementation is to carefully block data for the cache hierarchy of the processor as well as the register file in the machine. This optimization accounts for the majority of the performance gain in this application for a single processor and will be explored further in the next chapter.

Several new architectures choose to directly expose the memory hierarchy instead of emulating the traditional von Neumann architecture. For example, graphics processors (GPUs) and the STI Cell Broadband engine (Cell) require explicit movement of data into memories visible to the processor. In the case of GPUs, data must be moved from node memory into the graphics memory on the accelerator board for algorithm correctness. The SPEs in the Cell processor can only directly reference data in their small local memories, and data must be explicitly DMAed in and out of these memories during algorithm execution. Architectures that require explicit data movement are referred to as *exposed communication architectures* in the literature. Programming models that depend on a single address space fail to map efficiently to these architectures. Carefully using the memory system is no longer just about performance optimization but is also required for correctness on these architectures.

For both cache based and exposed communication memory systems, accessing data in bulk is required to efficiently use the memory hierarchy, and over time bulk access only becomes more important because latencies are rapidly increasing with respect to processing speed. In exposed communication memory systems, bulk data access translates into bulk data transfers. Since just initiating a transfer can have a latency of thousands of cycles, it is wise to transfer as much data as possible for each initiated transfer to amortize the transfer cost. For example, the cost of issuing a DMA for a single byte from host memory into the local store of a SPE on the Cell processor has the same latency as a 1KB data transfer. In a cache hierarchy based systems, accessing data in bulk leads to spatial locality in the cache, more efficient cache line prefetching, minimal misses to higher levels of the memory hierarchy which have even more access latency, and the amortization of cache miss costs along the cache line. Furthermore, some architectures like GPUs achieve extremely high bandwidths by using high latency but wide memory interfaces. For example, AMD's R600 processor uses a 512-bit memory interface to achieve greater than 100 GB/s to graphics memory [AMD, 2007]. However, this performance requires a burst size of 256-bytes for each memory request to efficiently make use of the wide interface and the high degree of interleaving and banking in the memory system.

Ideally, we would like to maximize the computational and bandwidth utilization of our machine. If we can overlap computation and communication, we can maximize the use of both for a given application. On cache machines, this can be done with data prefetching; exposed communication memory systems can use asynchronous transfer mechanisms. For optimal performance, we want to prevent stalling the compute resources as much as possible. As such, we need to do our best to make sure the data is available before computation begins. This requires identifying the data that will be needed next and starting transfer of

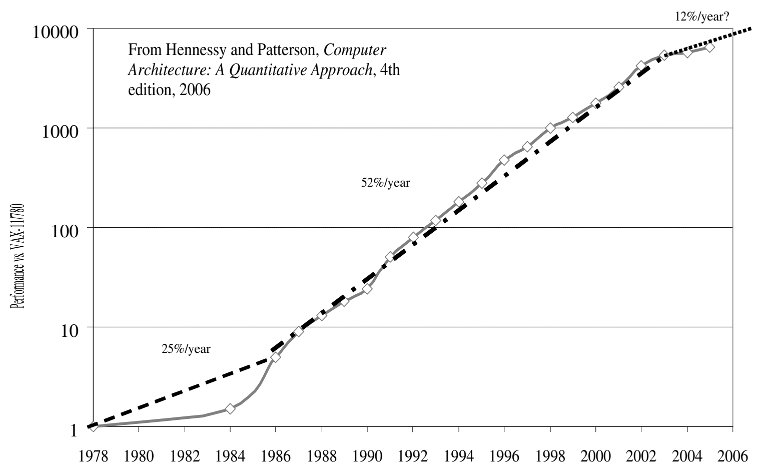

Figure 2.1: Single threaded scaling performance has come to an end.

the data as early as possible in the algorithm. This formulation is sometimes referred to as a *streaming* formulation. In practice, either computational resources or bandwidth resources become the limiting performance factor in this style of computation.

2.1.2 From Sequential to Parallel

As can be seen in Figure 2.1, performance scaling of single core performance according to Moore's Law has slowed considerably. Whereas single threaded performance has scaled at $\sim$52% per year from the mid-1980s to 2001, Intel now projects only a $\sim$10% performance increase each year in single threaded performance. Previous generations of processors have relied on progressively more advanced out-of-order logic, speculative execution, and superscalar designs along with increasing clock frequencies to continually increase performance of sequential, single threaded application performance. However, because of power and design limitations, we have largely hit the wall in scaling clock frequency, and superscaler processor design has reached the limit of available instruction level parallelism for most programs. Future processor designs are shifting transistor resources into multiple simplified processors on a single die. The STI Cell is a somewhat extreme example as the cores are in-order, have no branch prediction hardware, and the simplest cores on the die, the SPEs, do not even contain support for caches and require explicit data movement. In many ways, the design of each core represents the state of the art in architecture from more than a decade ago, albeit at much higher clock frequencies. The upcoming Intel Larabee design is comprised of many massively simplified x86 cores [Carmean, 2007]. Simpler cores allow a much denser packing of compute resources. Doug Carmean, the architecture lead of the Larabee project, has proposed that four of these simpler cores can fit in the same space as a current Core2 generation core. Clock for clock, the Larabee cores also have four times the theoretical compute performance of the traditional x86 designs from Intel, but sequential, single threaded performance may be as low as 30% of the current Intel designs [Carmean, 2007]. The difficulty with this architecture trend is that for programmers to increase application performance, they can no longer rely on improvements in sequential performance scaling and they now have to be able to effectively use parallel resources.

Traditional parallel programming techniques are beginning to break down as systems are becoming more and more parallel. The latest shipping CPUs currently have four cores, but road-maps from the CPU vendors show that scaling is expected to continue at a rate matching Moore's Law, meaning that if this scaling holds, then consumers will see 64 cores by 2015 and upwards of 100 cores in high-end workstation machines. In the high performance computing space, supercomputers have become extremely large, with the top 10 supercomputers having more than eight thousand processors. Ideally, we would like to have programming solutions that allow parallelism to scale easily from small numbers of processing elements to many, allowing algorithms and applications designed today on several cores to scale up to many cores.

2.2 Programming Systems

There has been a great deal of research on parallel languages, with some efforts going back several decades. Parallel languages and programming have continued to come in and out of vogue, with the last major efforts being in the mid to late 1990s. Most languages have focused on the high performance computing (HPC) domain, e.g. scientific computing like that performed at the US Department of Energy. High-performance computing has gradually become more common place with more industries now relying on large numbers of processors for financial modeling, bioinformatics, simulation, etc. As people have begun to see the reality of scalar processor performance hitting a wall, exploring programming models and parallel programming in general are actively being researched again. The DARPA HPCS program is now funding several research programming systems that reduce the cost and programming difficulty, increase the performance on large machines, provide portability across systems, and increase robustness of large applications [DARPA, 2007]. GPUs have also driven research into stream programming and data parallel languages in order to efficiently use these high performance, but esoteric, architectures [Owens et al., 2008].

While there have been many languages for parallel computing, most applications in HPC rely on MPI for distributed memory machines and OpenMP for shared memory systems. In the mainstream computing/consumer space, threading APIs like PThreads remain the most common. Streaming languages, largely driven by the difficulty in programming GPUs for more general computation than just graphics, remain largely ignored by the programming community and have yet to be used in common consumer applications or code development. Each language closely matches the underlying architecture it was originally targeted for, making portability to different machines while maintaining performance challenging.

2.2.1 Programming Models

Random Access Machine

The Random Access Machine (RAM) model [Aho et al., 1974] views a machine as a processor attached to a uniform and equal access cost memory system. A RAM is a multiple register machine with indirect addressing. Data access in a RAM program is modeled as instantaneous; thus, a processor never waits on memory references. The Parallel Random Access Machine (PRAM) model [Fortune and Wyllie, 1978] extends the RAM model to parallel machines. In a PRAM machine, data access from all processors to memory as well as synchronization between processors is modeled as instantaneous.

The RAM and PRAM models do not accurately model modern architectures. Even in a sequential system, data in the L1 cache can be accessed much faster than data in main memory, but all data transfers have some cost in terms of latency and are not instantaneous. Moreover, since data is transferred in bulk in modern architectures (cache-lines, memory-pages, etc.) locality of access is not taken into account in this model. Fine-grain, random data access is much more costly than bulk, coherent access because you can cause the memory system to load data in bulk (e.g. a cache-line of data) and then only use a small amount of the data loaded (e.g. a single byte of the cache-line). Despite only needing a small amount of data, the programmer pays for the bandwidth and latency of the larger transfer and can cause thrashing in the memory system. In the case of a parallel system, coherence protocols and non-uniform memory access (NUMA) designs further increase the cost of poor data access patterns. Moreover, the PRAM model treats synchronization as instantaneous and having no cost, but on modern architectures synchronization can cost hundreds of cycles. Algorithms designed using these computational models tend to perform poorly on modern architectures.

However, the RAM and PRAM model provide a very simple abstraction for computation and the PRAM model aides in understanding concurrency. These models serve as good teaching tools for algorithms and computational complexity, but provide little insight into the most performance critical aspects of contemporary algorithm design and analysis.

Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) model [Valiant, 1990] differs from the PRAM model in that communication and synchronization costs are not assumed to be free. An important part of the analysis of programs written according to the BSP model is the quantification of the communication and synchronization during execution. A BSP program is comprised of three supersteps: 1) concurrent computation, 2) communication, and 3) synchronization. During the computation phase, the same computation occurs independently on all processors operating only on data local to each processor. During the communication phase, the processors exchange data between themselves en masse. During the synchronization phase, each processor waits for all other processors to complete their communication phase. Algorithms are comprised of many of these supersteps.

The main advantages of BSP over PRAM is that algorithms are comprised of separate computation, bulk communication, and synchronization steps. The programmer is made aware of the cost of communication and synchronization and is encouraged to transmit data in larger chunks. However, the BSP model as presented in the literature does not model memory hierarchies with more than two levels: main memory and processor memory.

LogP

The LogP model [Culler et al., 1993] is based on parameters that describe the latency (L), communication overhead (o), gap between consecutive communications (g), and the number of processor/memory models (P) of the machine. Compared to the BSP mode, the LogP model has more constrained communication mechanisms and lacks explicit synchronization, but allows for more flexible communication and execution capabilities. Unlike the BSP model, communication and computation are asynchronous, and a processor can use a message as soon as it arrives, not just at superstep boundaries. LogP works to encourage coordinating the assignment of work with data placement to reduce bandwidth requirements as well as encouraging algorithms that overlap computation and communication within the limits of network capacity.

Like BSP, LogP does not assume zero communication delay or infinite bandwidth, nor does it tailor itself to a specific interconnect topology as do simpler models. Implicit in the model is that processors are improving in performance faster than interconnect performance, and that latency, communication overhead, and limited bandwidths will be the performance critical aspects of algorithm design. However, like the BSP model, the LogP model models two level memory hierarchies and is really targeted towards modeling interconnected computers and not the full memory hierarchy of the machine. Some researchers have argued that the BSP model is a more convenient programming abstraction and computational model for parallel computation; however, the LogP model can be more exact in modeling some machines as compared to BSP [Bilardi et al., 1996].

Cache Oblivious

Cache oblivious algorithms are designed to exploit a cache hierarchy without knowledge of the specifics of the caches (number, sizes, length of cache lines, etc.) [Frigo et al., 1999]. In practice, cache oblivious algorithms are written in a divide-and-conquer form where the problem is progressively divided into smaller and smaller sub-problems. Eventually, the sub-problem will become small enough to fit in a cache level and further division will fit the problem into smaller caches. For example, matrix multiply is performed by recursively dividing each matrix into four parts and multiplying the submatrices in a depth first manner. However, for optimal code, the user must define a base case for the recursion that allows for an efficient implementation of the computation. In practice, the base case stops recursion after the data fits in the cache closest to the processor and an optimized function is written to optimize register usage and take advantage of available SIMD instructions. The elegance of cache oblivious algorithms is that they can make efficient use of the memory hierarchy in an easy to understand way. Accordingly, this is an attractive place to start the design of our system.

However, there are several issues with the cache oblivious approach. Firstly, the cache oblivious model makes the assumption that the memories are caches and that data at the base of the recursion can be accessed via global addresses. This model has problems on systems that have exposed communication hierarchies as the address spaces are distinct and cannot be accessed using global addresses. The cache oblivious model also relies on a memory system comprised of a cache hierarchy in which all data access can be driven from the bottom of the hierarchy and misses into a cache will generate requests into the cache above it and so on until the memory memory request can be satisfied and the data can be pulled into the lowest level cache. It also assumes that higher level caches are inclusive, i.e. they include all of the data in the caches below them. We must stall on every miss and rely on low miss rates, which is the general case for optimal cache oblivious algorithms. However, since the data access is fine grained and generated from the bottom of the memory system, we do not have the ability to transfer data in bulk, required for efficient memory transfers on exposed communication hierarchies, nor the ability to prefetch data to avoid stalls and allow for overlapping computation and communication. Parallelism is also not able to be directly described in this model.

Streaming

The stream programming model is designed to directly capture computational and data locality. A *stream* is a collection of records requiring similar computation while *kernels* are functions applied to each element of the stream. A streaming processor executes a kernel for each element of the input stream(s) and places the results into the output stream(s). Similar to BSP, streaming computations are comprised of a communication phase to read inputs, a computation phase performing calculations across the inputs, and a communication phase placing the results into the outputs. However, unlike BSP, synchronization is not required as each stream element is executed on independently and communication does not exist between elements. Streaming formulations also have the benefit that the communication and computation phases are overlapped to maximize the resources of the machine. Also, stream programming encourages the creation of applications with high arithmetic intensity, the ratio of arithmetic operations to memory bandwidth [Dally et al., 2003], with the separation of computation into kernels applied to streams. The drawback of traditional streaming approaches is they only handle two levels of the memory hierarchy, off processor and on processor memory.

2.2.2 Programming Languages

Global View

Global view languages allow for arbitrary access to the total system memory, providing RAM/PRAM models of computation. ZPL [Deitz et al., 2004], Chapel [Callahan et al., 2004], and High Performance Fortran (HPF) [Forum, 1993] are examples of global view languages. These three languages are array-based, borrowing from Fortran syntax. Since there are no pointers or pointer arithmetic, more aggressive compiler analysis is possible. All three languages allow the programmer to express parallelism via simple data parallel iteration constructs such as `parallel for` loops. The data parallel basis of these languages provides guarantees that there is no aliasing on writes to arrays and execution and control are defined in bulk, which allows for aggressive optimizations. Since synchronization primitives are not exposed to the user and there are guarantees about data aliasing, the compiler can be aggressive in scheduling by relying on data dependence analysis of the call chain.

A limitation of these languages is that there is no first class language support for specifying how to place data in the machine for locality. *Regions* combined with *distributions* in ZPL and Chapel provide information about what data is needed for a computation but not where that data resides. Regions do have the nice property that the area of memory that can be accessed is well defined so that memory movement can be scheduled in the case of distributed memory, but these languages do not provide a way to specify how data should be laid out to minimize communication. Furthermore, these languages have no notion of the memory hierarchy of the machine on which the code is executing and allow fine-grain data access to the global system memory. As mentioned above with the RAM/PRAM models, this can lead to inefficient data access. In theory, the region constructs from ZPL/Chapel could be nested in order to decompose the data, but the user would have to explicitly manage the decomposition for each target machine.

Partitioned Global Address

Partitioned Global Address Space (PGAS) languages have been the most successful parallel programming languages to date in terms of implementations available on many machines and user community size. For example, Unified Parallel C (UPC) [Carlson et al., 1999] is available on several supercomputers including the Cray X1 family and T3E series, as well as large cluster machines such as Blue Gene/L. Co-Array Fortran [Numrich and Reid, 1998], an extension of Fortran 95, is being studied for inclusion into the Fortran 2008 specification. Titanium [Yelick et al., 1998] is a Java-based language with similar properties to UPC.

The PGAS model presents two levels of the memory hierarchy: data is either *local* to a processor or *global* to all processors. PGAS languages can only capture locality in two levels of memory, limiting execution to architectures which can be abstracted as two level machines. The PGAS model also allow fine grain data access, making the generation of bulk data transfers difficult, thus leading to potentially inefficient execution on architectures such as distributed memory systems. Programmers specify whether data is local or global and can access each in standard C syntax. UPC has recently gained API functions to support bulk transfers via `memcpy`'s to help improve performance across slower interconnects; however, the user must decide when they are going to use bulk transfers over fine grain data access. Similar extensions have been proposed for Titanium and Co-Array Fortran. The added performance of using these extended mechanisms come at the cost of portability as not all machines nor compiler/runtime implementations have support for these constructs.

In UPC 2.0, extensions for asynchronous bulk transfers have also been proposed, but the user is responsible for scheduling data movement.

PGAS models also have difficulty with memory coherence and consistency on machines that do not provide these capabilities in hardware. Since the machine is presented to the user as distributed shared memory, the user is responsible for synchronization, but memory consistency behavior is unclear in the general case ("relaxed consistency" in the UPC documentation) and behavior can vary between machine types. This leads to overzealous synchronization in user code, possibly sequentializing execution during large parts of the code. Another subtle issue with the current PGAS models is that they do not support nested-parallelism. As such, the nested parallel loops must be flattened into a single parallel loop by the user manually to increase parallelism.

Threading languages

Cilk [Blumofe et al., 1995] provides a language and runtime system for light-weight threading, which is particularly suited, but not limited, to cache-oblivious algorithms implicitly capable of using a hierarchy of memories. Cilk is a simple addition the the C programming language, and simple elision of Cilk programs can be compiled by standard C compilers and executed on sequential machines. The main difference between Cilk and C is the support for a fork/join execution model in Cilk. The user can spawn threads for computation using fork and then wait for thread completion with joins. Cilk relies on an efficient runtime system to spawn and schedule threads to processing elements.

Like C, the user is allowed to use fine grain memory access from global data and may run into the performance pitfalls of a random access memory model. Moreover, compiler optimizations are made difficult by potential pointer aliasing in the programmer's code. When code is written in a cache-oblivious manner, the behavior of the memory system can be vastly improved, but access is still fundamentally to the entire global memory. This limits efficient execution of Cilk to shared memory machines. However, Cilk's runtime system has the ability to better handle irregular computations than many other systems.

GPGPU

Given the promising computational capabilities of graphics processors, there have been several academic and industry efforts to create languages for general purpose computation on graphics processors (GPGPU). BrookGPU [Buck et al., 2004], a derivative of the Brook [Ian Buck, 2003] streaming language based on C with streaming extensions, is designed to abstract a graphics processor as a streaming processor. Data is explicitly transferred to and from host memory using `streamRead` and `streamWrite` operators to initialize streams of data. As a streaming computation model, the user defines kernels which operate over streams. The kernels are invoked once per output stream element and executed in a data parallel fashion with no communication or synchronization between kernel invocations. BrookGPU allows kernels to read from streams in a general way (*gathers*) but does not allow arbitrary writes to streams (*scatter*). The functional capabilities of BrookGPU directly correspond to what can be done using graphics APIs, and all runtime calls and kernels are mapped to graphics API primitives such as textures, framebuffers, and fragment shaders. BrookGPU is inherently a two level memory model with explicit data transfers: data is either in host memory or in device memory. However, there are multiple levels of cache and scratch-pad memories available on the latest GPUs that are not exposed via the programming model that can limit application performance.

Nvidia's CUDA [NVIDIA, 2007] programming language is based on C with extensions for data parallel execution. CUDA presents the GPU as a bag of parallel processors on which programs can be executed. Similar to BrookGPU, the user must explicitly transfer data from host memory to device memory before program execution. However, CUDA does not use streaming semantics during program execution and instead uses explicit general reads and writes via pointers and arrays. The user describes an execution grid which specifies how many times to invoke the program in total creating the specified number of threads, and how to divide the execution grid into blocks of threads to be scheduled on the processor. Moreover, CUDA exposes small scratch-pads per processor that data can be explicitly read to and written from the program to allow data sharing between threads on a processor. This memory is not directly in a hierarchy, i.e. one cannot cause a transfer from device memory directly to the scratch-pad and instead must use registers as an intermediate and use two transfer operations. Thus, CUDA exposes three memories to the user– host, device, and scratch-pad. Another difference between CUDA and BrookGPU is that CUDA allows limited synchronization. Synchronization is defined for threads in a block allowing for data sharing and communication between threads via the scratch-pads. While there are claims of performance gains using this model, the user must explicitly code to the specific architecture implementations to use these more advanced features potentially limiting portability to other GPUs.

2.2.3 Runtime Systems and APIs

Compiler assisted

OpenMP is a successful system for parallelizing code via compiler hints on shared memory machines. Programmers write their code in standard programming languages like C, C++, and Fortran, and provide hints to the compiler via pragmas about which loops can be parallelized and how the execution of the loop should be distributed among processors. More aggressive compilers will attempt to automatically parallelize all loops. OpenMP is very attractive to programmers because they do not have to use a new programming language or model and still get parallel code. The compilers must be conservative in parallelizing to maintain correctness but often struggle with pointer aliasing. The user must progressively add hints and/or re-factor their code to avoid the potential for aliasing and help expose more potential parallelism to the compiler in order to gain better performance. Unfortunately, the pragmas differ between compilers, although there is a standard subset which is generally used. Using the vendor specific extensions for better optimization and targeting of a specific machine come at the cost of portability. Moreover, in practice, the ability of OpenMP compilers to parallelize arbitrary code is very limited and users have to go through many iterations exploring pragmas and restructuring their code to allow the compiler to better parallelize.

OpenMP can be inefficient on large shared memory machines because of non-uniform memory access effects and makes distributed memory implementations problematic since there is no notion of locality and the model is built around fine-grain global data access. Since there are no direct methods for bulk data transfers, performance can suffer greatly across slow interconnects, or the user must reorder their code and add progressively more hints to get the compiler to generate code for efficient data transfers. OpenMP is also unable to provide asynchronous data transfers thus leading to a reactive memory system in the common case where a data access may cause a stall for an undefined amount of time. Some implementations can work around this by implementing user level threads on each execution unit and switching execution threads on expensive data transfers.

APIs

The Parallel Virtual Machine (PVM) [Geist et al., 1994] and MPI [MPIF, 1994], both preceded by [Su et al., 1985], are perhaps the oldest and most widely used systems for programming parallel machines and are supported on many platforms. Both systems concentrate on the explicit movement of data between processors within one logical level of the machine. The user must specify all communication manually and communication requires both the sending and receiving node to be involved. The user must also explicitly control the creation of parallel contexts and all synchronization. MPI-2 [MPIF, 1996] adds support for single sided data transfer making programming easier, but these functions are not supported on all platforms. MPI-2 can also abstract parallel I/O resources, thus exposing another memory level, but the API is very different from the core communication API functions.

The Pthreads library allows direct programming of shared-memory systems through a threading model and also assumes a uniform global address space. The user is responsible for creation, destruction, and synchronization of all parallel contexts. Moreover, since by default all data in the parent process is shared by all threads, so the user is responsible for maintaining thread local storage and managing communication and synchronization between threads. Other two-level runtime systems include Charm++ [Kalé and Krishnan, 1993], Chores [Eager and Jahorjan, 1993], and the Stream Virtual Machine (SVM) [Labonte et al., 2004]. None of these systems are designed for handling more than two levels of memory or parallel execution in a unified way.

The IBM Cell SDK [IBM, 2007b] provides an API for programming for the IBM Cell processor. This API is a very low level programming interface closely matching the hardware and explicitly supports only two-levels of memory. The user must create and manage executions contexts on the SPEs, manage loading of executable code into the SPEs as overlays, communicate between the PPE and SPE, and control DMAs and synchronization. This API is unlike any of others described and provides yet another distinct programming system and mindset.

Chapter 3

Abstract Machine Model

In order to meet our goals of portability across machines while maintaining good performance, we need to find a computational model and machine abstraction that fits our needs. As discussed previously, modern architectures are gaining ever increasing amounts of parallelism (Section 2.1.2) and deeper memory hierarchies (Section 2.1.1). As such, we need to find a computational model that encapsulates the performance critical aspects of modern architectures. Also, since there are many different types of architectures, we need to find a uniform way to abstract machines.

Most theoretical machine models in computer science do not address certain performance issues important for creating high performance programs on modern architectures. Careful tuning of an algorithm to closely match the characteristics of the architecture can lead to more than an order of magnitude increase in program performance. Many performance tuning problems that arise after the algorithm and data structures have been chosen amount to efficiently moving data through the machine. Much of the large performance increase comes from taking into account the various aspects of the memory hierarchy of the target machine. However, this tuning requires detailed knowledge of the machine's architectural features.

Traditional models of computation, such as the Random Access Machine (RAM), ignore the non-uniform cost of memory access. For example, let us explore the performance of matrix multiplication to show the difference between theory and practice. In the RAM model, where every memory access is uniform, the complexity of this program is $O(N^3)$. Implementing the matrix multiplication in this model leads to the traditional naive triple-nested `for` loop formulation. However, real contemporary systems have multiple levels of caching and requirements on alignment for performance and this formulation which are not well suited to this formulation. On an Intel 2.4 GHz Pentium4 Xeon machine, this naive implementation compiled with the Intel compiler performs a 1024x1024 matrix multiplication at $1/150$th the performance of Intel's Math Kernel Library, more than two orders of magnitude lower performance. The performance of the naive implementation is limited by the latency and throughput of the last level of the memory hierarchy. However, if we take into account the memory hierarchy of this machine, we can greatly increase performance to the point where the actual performance better matches the performance of a highly tuned implementation.

The Intel Pentium4 Xeon processor has several levels of memory hierarchy as shown in Figure 3.1: a register file, a 32 KB L1 cache, a 512 KB L2 cache, and main memory. The register file is extremely fast but very small. As we get further away from the functional units, the larger the memory gets, but the lower the bandwidth and the higher latency becomes. We can start by first performing small blocked matrix multiply operations in closest/fastest memory, the register file, and then building the larger matrix multiply in terms of these smaller matrix multiplies. This leads to a 6-nested `for` loop implementation, a triple-nested `for` loop representing the 4x4 matrix multiplies fitting into the registers, and another triple-nested `for` loop which blocks the full matrix multiplication in terms of the

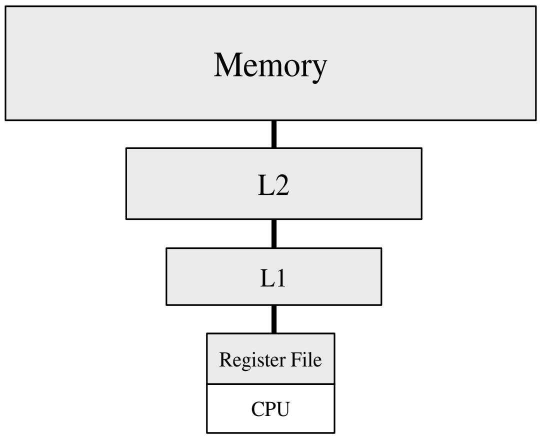

Figure 3.1: Intel Pentium4 Uniform Memory Hierarchy example

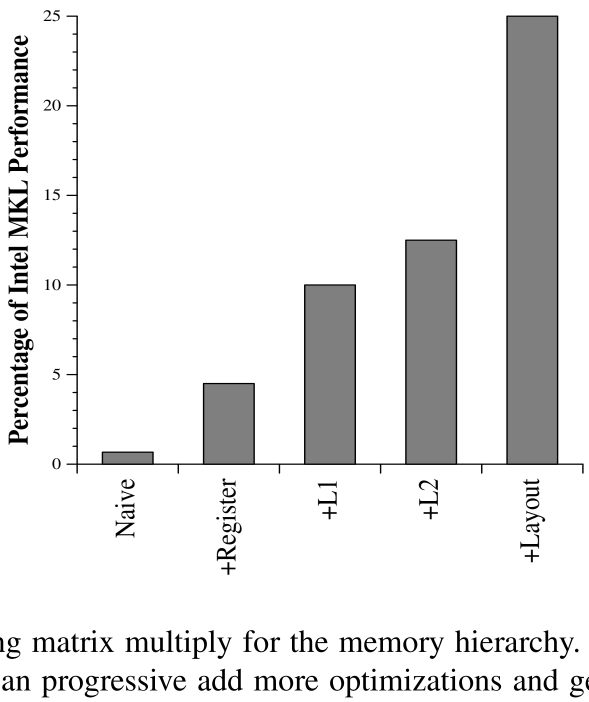

Figure 3.2: Optimizing matrix multiply for the memory hierarchy. Starting from a naive implementation, we can progressive add more optimizations and get to within 1/4 of the performance of the highly tuned MKL library with only memory system optimizations.

smaller 4x4 matrix multiplications. This adaptation to just explicitly take advantage of the fastest/lowest level of the memory hierarchy, without considering any other levels explicitly, increases performance to $1/22$ of MKL, a performance increase of almost a factor of 7 with just this simple modification. Similarly, we can block for execution into the next level of the memory hierarchy, the L1 cache, performing 32x32 matrix multiplies comprised of 4x4 register matrix multiplies, by adding yet another set of triple-nested `for` loops. Blocking for the L1 cache increases performance by another factor of 2, to $1/10$ of MKL performance. Blocking again for the next level of the memory hierarchy, the L2 cache, performing 256x256 matrix multiplies, increases performance to $\sim 1/8$ of MKL performance. If we manually reformat the data to better match the cache line sizes of the processor and the data order in which the hardware prefetch units function, we can get within $1/4$ of the performance of MKL. The performance effect of this progression of optimization is shown in Figure 3.2. Notice that we can achieve $1/4$ of the performance of a processor vendor's code with only optimizations for the memory hierarchy. Hand-tuning the inner loop along with fairly heroic optimizations yields the remaining factor of 4 in performance.

3.1 Memory Hierarchy Model

The matrix multiplication example motivates a programming model that captures the relevant performance aspects of the hierarchical nature of computer memory systems. The Uniform Memory Hierarchy (UMH) model of computation [Alpern et al., 1994] presents a framework for machines with more than two levels in the memory hierarchy. As a theoretical model, UMH refines traditional methods of algorithm analysis by including the cost of data movement through the memory hierarchy. However, the UMH model also provides a way to abstract machines as a sequence $\langle M_0, \ldots, M_n \rangle$ of increasingly larger memory modules with computation taking place in M_0. For example, M_0 may model the computer's central processor and register file, while M_1 might be cache memory, M_2 main memory, and so on including all levels of the memory hierarchy in a given machine. For each module M_n, a bus B_n connects it with the next larger memory module M_{n+1}. Buses between multiple memory levels may be active, simultaneously transferring data. Data is transferred along a bus in fixed-sized blocks. The size of these blocks, the time required to transfer a block, and the number of blocks that fit in a memory module increase as one moves up the memory hierarchy.

An important performance feature of the UMH model is that data transfers between multiple memory modules in the hierarchy can be active simultaneously. Hence, the UMH model accounts for overlapping computation and communication, leading to programs bound by memory performance or compute performance rather than the sum of communication and computation. The UMH model expresses the tight control over data movement and the memory hierarchy that is necessary for achieving good performance on modern architectures.

As mentioned in Section 2.1.2, we are quickly moving away from sequential processing to parallel processing. While the UMH model has been shown to be a good match for performance programming on sequential processors, it does not provide a solution for parallel processors. The Parallel Memory Hierarchy (PMH) model [Alpern et al., 1993] extends the UMH model for parallel systems. Instead of a linear connection of memory modules, the PMH model abstracts parallel machines as a tree of memory modules, see Figures 3.3 and 3.4. Similar to the UMH model's benefits over the RAM model of computation, the PMH model provides a better computational model than the Parallel Random Access Machine (PRAM) model. The PRAM model is a special case of the PMH model with only two levels of memory: a root memory module representing all of memory with p children each having a memory of size 1. The use of a tree to model a parallel computer's communication

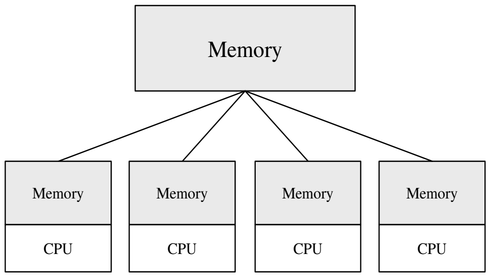

Figure 3.3: Two level example of a Parallel Memory Hierarchy

structure is a compromise between the simplicity of the PRAM model and the accuracy of a arbitrary graph structure.

The PMH abstract representation of a system containing a Cell processor (at left in Figure 3.5) contains nodes corresponding to main system memory and each of the 256KB software-managed local stores (LSes) located within the chip's synergistic processing units (SPEs). At right in Figure 3.5, a PMH model of a dual-CPU workstation contains nodes representing the memory shared between the two CPU's as well as the L1 and L2 caches on each processor. The model permits a machine to be modeled with detail commensurate with the programmer's needs. A representation may include modules corresponding to all physical levels of the machine memory hierarchy, or it may omit levels of the physical hierarchy that need not be considered for software correctness or performance optimization.

3.2 Space Limited Procedures

Space Limited Procedures (SLP) [Alpern et al., 1995] provides a methodology for programming in the PMH model and defines the general attributes of the underlying system.

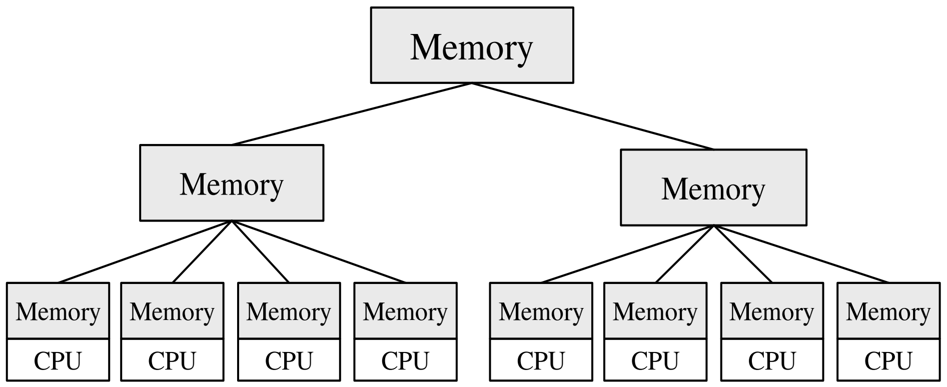

Figure 3.4: Three level example of a Parallel Memory Hierarchy

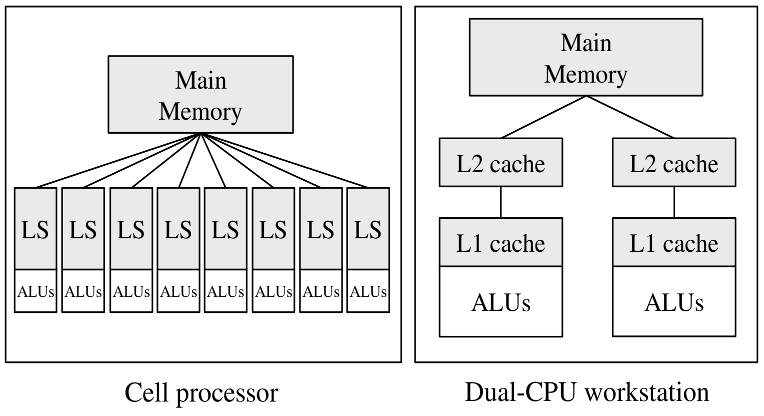

Figure 3.5: A Cell workstation (left) is modeled as a tree containing nodes corresponding to main system memory and each of the processor's software-managed local stores. A representation of a dual-CPU workstation is shown at right.

SLP takes the PMH model and transforms the theoretical model into a methodology for obtaining portable high-performance applications.

To achieve high performance on a machine, the processing elements of that machine must be kept as busy as possible doing useful work. To keep processing elements busy, one must:

- decompose problems into independent sub-problems that can be executed concurrently,
- distribute these sub-problems among the processing elements in the machine,
- place necessary data as close to the processing element as possible, and
- overlap communication with computation when possible.

In SLP, each memory module can hold at least as much data as all its children combined and is parameterized by its capacity and the number of children it has. Data moves between a module and a child over a channel (bus) in fixed sized blocks. Each memory module runs a PMH routine that choreographs the flow of data between a module and its children and invokes PMH routines on its children. A problem instance begins in a memory module that is large enough to satisfy the application's storage requirements. The problem is then broken into sub-problems that can be executed using the storage available in the current module's children. Before a routine is invoked on the child, the input data must be present in the child memory module as well as storage available for the routine's results. Thus, the parent transfers data to its children, starts sub-problems in the children, waits for completion, and transfers the results back. These sub-problems are broken down further into progressively smaller sub-problems and passed down the tree of memories. Eventually, sub-problems small enough to fit into the leaves flow into the leaves where they are solved and their results are returned up the tree. The solutions to sub-problems may either be used as input to later sub-problems or passed up to the parent as part of the solution of the current problem.

SLP programs are comprised of procedures that call procedures. The resulting call graph structure directly reflects the PMH tree structure representation of the target machine. Calls that can be executed in parallel may be identified explicitly or deduced via analysis. Alternative algorithms and/or data structures are indicated by overloading procedure names, thus providing multiple variants of the same procedure. Tuning parameters for space limited programs come in three forms: *machine parameters* are the parameters of the PMH model and reflect the performance relevant features of the target computer, *problem parameters* reflect the performance relevant features of the problem instances, and *free parameters* which defer other performance relevant choices until program specialization. Arguments to a procedure are identified as `read`, `write`, or `readwrite`.

The body of a variant of a space-limited procedure is composed of conventional programming language statements: control structures, procedure calls, and assignments. Arguments to the called procedure must take up substantially less space than those of the calling procedure. During specialization, the tuning parameters are defined, variants are chosen, and procedure calls are mapped to different memory modules in the machine.

3.3 The Sequoia Model

Although the UMH and PMH work define a theoretical model of computation and the SLP work provides a programming methodology, they do not provide an explicit abstraction nor an implementation for any parallel machines. We need a way to provide a uniform abstraction that allows us to efficiently execute on many parallel architectures, and then develop a language, compiler, and runtime system around this abstraction.

We adapt the SLP methodology with the realization that the techniques required to accommodate the different mechanisms in different levels of the memory hierarchy, from

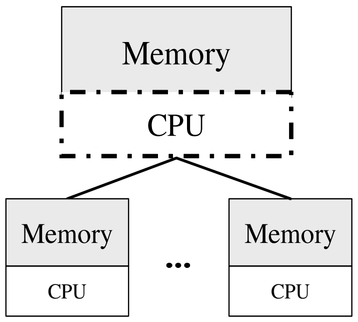

Figure 3.6: Two level example of our abstraction. Each tree node is comprised of a control processor and a memory. Interior control processors, denoted with a dashed line, can only operate to move data and transfer control to children. Leaf control processors are also responsible for executing user defined procedures

network interfaces to disk systems, are fundamentally the same. We model the diverse features among different systems with the same mechanism, the *tree node*, and the capabilities we allow for the tree nodes. From Section 2.1.1, we know the importance of the memory hierarchy and that bulk asynchronous transfers are required for performance on many machines. From Section 2.1.2, we know that we need to accommodate parallelism explicitly.

Since many machines have complex non-tree topologies, we allow our tree abstraction to include *virtual levels* that do not correspond to any single physical machine memory. For example, it is not practical to expect the nodes in a cluster of workstations to communicate only via global storage provided by networked disk. As shown in Figure 3.7, our model represents a cluster as a tree rooted by a virtual level corresponding to the aggregation of all workstation memories. The virtual level constitutes a unique address space distinct from any node memory, e.g. memory designated as part of the virtual level does not overlap with memory designated for the node level. Transferring data from this global address space into the child modules associated with individual cluster workstations results in communication over the cluster interconnect. The virtual level mechanism allows us to generalize the

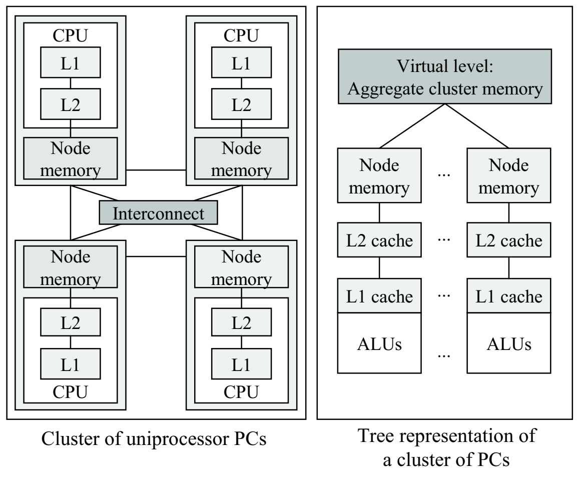
CPU
L1
L2
Node memory
Interconnect
Virtual level: Aggregate cluster memory
L2 cache
L1 cache
ALUs
...
Cluster of uniprocessor PCs
Tree representation of a cluster of PCs

Figure 3.7: The point-to-point links connecting PCs in a cluster are modeled as a virtual node in the tree representation of the machine.

tree abstraction for modeling vertical communication to encapsulate horizontal inter-node communication as well. The virtual level is analogous to the PGAS model's global memory and the actual physical node memory analogous to the PGAS model's the local memory.

Following from the PMH model, we abstract machines as a tree of nodes. We formalize the abstraction as follows:

A node has one *control thread* and one memory.

Threads can:

- transfer in bulk to/from parent asynchronously,
- wait for transfers to/from parent to complete,
- allocate data in their memory,
- only access their memory directly,
- transfer control to child nodes,

- wait for children to complete execution,
- synchronize with siblings, and
- non-leaf threads can only operate to move data and transfer control.

The main differences between the implied SLP abstraction and ours are child centric data transfers and sibling synchronization. The original SLP work states that transfers happen from parent to child before the child begins execution. On modern parallel systems, this formulation creates several problems. As stated above, in order to model a cluster of workstations, we have included the notion of virtual levels to represent the distributed aggregate memory of the cluster. If we allowed parent driven transfers, this would mean that a virtual node would have to transfer data to each of its children, represented by actual machines. Since the node is virtual, it does not actually own any data making parent directed transfers unnatural. If instead we initiate transfers from the children, transfers with the parent turn into horizontal communication with other nodes. On machines without virtual levels, initiating transfers with the children instead of the parent allows for distributed communication, thus improving data transfer performance.

In the original SLP work, synchronization was performed by returning control to the parent node. When control is returned to the parent, the output data from the child execution must also be transferred to the parent. However, if there is significant data reuse in a child between procedures but the next procedure needs data computed by a sibling and returned to the parent, then we can save overhead and potentially extra data transfers if we can synchronize siblings without returning control to the parent.

This abstraction allows us to capture the performance critical aspects of machines, including the use of parallel resources and efficient use of the memory hierarchy while providing a practical match to actual machine capabilities.

Chapter 4

Sequoia

4.1 Hierarchical Memory

In Figure 4.1, we illustrate the hierarchical structure of a computation to perform blocked matrix multiplication, an example we will revisit in this chapter and is the example we used to motivate the Sequoia machine model in Chapter 3. In this algorithm, which features nested parallelism and a high degree of hierarchical data locality, parallel evaluation

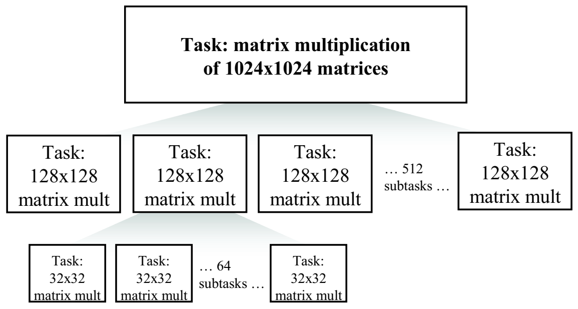
Task: matrix multiplication of 1024x1024 matrices
Task: 128x128 matrix mult
Task: 128x128 matrix mult
Task: 128x128 matrix mult
… 512 subtasks …
Task: 128x128 matrix mult
Task: 32x32 matrix mult
Task: 32x32 matrix mult
… 64 subtasks …
Task: 32x32 matrix mult

Figure 4.1: Multiplication of 1024x1024 matrices structured as a hierarchy of independent tasks performing smaller multiplications.

of submatrix multiplications is performed to compute the product of two large matrices. Sequoia requires such hierarchical organization in programs, borrowing from the idea of Space Limited Procedures (Section 3.2), to encourage hierarchy-aware, parallel divide-and-conquer programs. Sequoia tasks (Section 4.2) generalize and make concrete the concept of a space-limited procedure into a central construct used to express communication and parallelism and enhance the portability of algorithms. We have implemented a complete programming system around this abstraction.

Writing Sequoia programs involves abstractly describing hierarchies of tasks (as in Figure 4.1) and then mapping these hierarchies to the memory system of a target machine. Sequoia requires the programmer to reason about a parallel machine as a tree of distinct memory modules, a representation that extends the Parallel Memory Hierarchy (PMH) model of Alpern et al. [1993] (Section 3.3). Data transfer between memory modules is conducted via (potentially asynchronous) block transfers. Program logic describes the transfers of data at all levels, but computational kernels are constrained to operate upon data located within leaf nodes of the machine tree.

Establishing an abstract notion of hierarchical memory is central to the Sequoia programming model. Sequoia code does not make explicit reference to particular machine hierarchy levels and it remains oblivious to the mechanisms used to move data between memory modules. For example, communication described in Sequoia may be implemented using a cache prefetch instruction, a DMA transfer, or an MPI message depending on the requirements of the target architecture. Supplying constructs to describe the movement of data throughout a machine while avoiding any reference to the specific mechanisms with which transfers are performed is essential to ensuring the portability of Sequoia programs while retaining the performance benefits of explicit communication.

As with the PMH model, our decision to represent machines as trees is motivated by the desire to maintain portability while minimizing programming complexity. A program that performs direct communication between sibling memories, such as a program written using MPI for a cluster, is not directly portable to a parallel platform where such channels do not exist.

4.2 Sequoia Language

The principal construct of the Sequoia programming model is a *task*: a side-effect free function with call-by-value-result parameter passing semantics. Tasks provide for the expression of:

- **Explicit Communication and Locality**. Communication of data through the memory hierarchy is expressed by passing arguments to tasks. Calling tasks is the only means of describing data movement in Sequoia.
- **Isolation and Parallelism**. Tasks operate entirely within their own private address space and have no mechanism to communicate with other tasks other than by calling subtasks and returning to a parent task. Task isolation facilitates portable concurrent programming.
- **Algorithmic Variants**. Sequoia allows the programmer to provide multiple implementations of a task and to specify which implementation to use based on the context in which the task is called.
- **Parameterization**. Tasks are expressed in a parameterized form to preserve independence from the constraints of any particular machine. Parameter values are chosen to tailor task execution to a specific hierarchy level of a target machine.

```
void task matmul::inner( in    float A[M][T],
                         in    float B[T][N],
                         inout float C[M][N] )
{
  // Tunable parameters specify the size
  // of subblocks of A, B, and C.
  tunable int P;
  tunable int Q;
  tunable int R;

  // Compute all blocks of C in parallel.
  mappar (int i=0 to M/P, int j=0 to N/R) {
    mapseq (int k=0 to T/Q) {
      // Invoke the matmul task recursively
      // on subblocks of A, B, and C.
      matmul(A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
             B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
             C[P*i:P*(i+1);P][R*j:R*(j+1);R]);
    }
  }
}

void task matmul::leaf( in    float A[M][T],
                        in    float B[T][N],
                        inout float C[M][N] )
{
  // Compute matrix product directly
  for (int i=0; i<M; i++)
    for (int j=0; j<N; j++)
      for (int k=0; k<T; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

Figure 4.2: Dense matrix multiplication in Sequoia. `matmul::inner` and `matmul::leaf` are variants of the `matmul` task.

This collection of properties allows programs written using tasks to be portable across machines without sacrificing the ability to tune for performance.

4.2.1 Explicit Communication And Locality

A Sequoia implementation of blocked matrix multiplication is given in Figure 4.2. The `matmul` task multiplies M x T input matrix `A` by T x N input matrix `B`, accumulating the results into M x N matrix `C` (`C` is a read-modify-write argument to the task). The task partitions the input matrices into blocks (lines 16–18) and iterates over submatrix multiplications performed on these blocks (lines 12–20). An explanation of the Sequoia constructs used to perform these operations is provided in the following subsections.

The definition of a task expresses both locality and communication in a program. While a task executes, its entire *working set* (the collection of all data the task can reference) must remain resident in a single node of the abstract machine tree. As a result, a task is said to run at a specific location in the machine. In Figure 4.2, the matrices `A`, `B`, and `C` constitute the working set of the `matmul` task. Pointers and references are not permitted within a task and therefore a task's working set is manifest in its definition.

Notice that the implementation of `matmul` makes a recursive call in line 16, providing subblocks of its input matrices as arguments in the call. To encapsulate communication, Sequoia tasks use *call-by-value-result* (CBVR) [Aho et al., 1986] parameter passing semantics. Each task executes in the isolation of its own private address space (Subsection 4.2.2) and upon task call, input data from the calling task's address space is copied into that of the callee task. Output argument data is copied back into the caller's address space when the call returns. The change in address space induced by the recursive `matmul` call is illustrated in Figure 4.3. The block of size P x Q of matrix `A` from the calling task's address space appears as a similarly sized array in the address space of the called subtask. CBVR is not common in modern languages, but we observe that for execution on machines where data is transferred between distinct physical memories under software control, CBVR is a natural parameter passing semantics.

The mapping of a Sequoia program dictates whether a callee task executes within the same memory module as its calling task or is assigned to a child (often smaller) memory module closer to a compute processor. In the latter case, the subtask's working set must be transferred between the two memory modules upon task call/return. Thus, the call/return of a

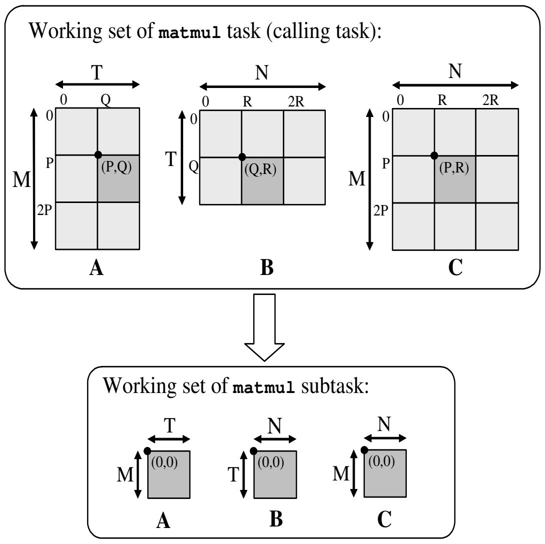
Working set of matmul task (calling task):
T
N
N
0
Q
R
2R
P
2P
M
(P,Q)
(Q,R)
(P,R)
A
B
C
Working set of matmul subtask:
(0,0)
A
B
C

Figure 4.3: The `matmul::inner` variant calls subtasks that perform submatrix multiplications. Blocks of the matrices *A*, *B*, and *C* are passed as arguments to these subtasks and appear as matrices in the address space of a subtask.

subtask implies that data movement through the machine hierarchy *might* occur. Explicitly defining working sets and limiting communication to CBVR parameter passing allows for efficient implementation via hardware block-transfer mechanisms and permits early initiation of transfers when arguments are known in advance.

4.2.2 Isolation and Parallelism

The granularity of parallelism in Sequoia is the task and parallel execution results from calling concurrent tasks. Lines 12–20 of Figure 4.2 describe iteration over submatrix multiplications that produces a collection of parallel subtasks. The `i` and `j` dimensions of the iteration space may be executed in parallel while the innermost dimension defines a sequential operation which performs a reduction. In Sequoia, each of these subtasks executes in isolation, which is a key property introduced to increase code portability and performance.

Isolation of task address spaces implies that no constraints exist on whether a subtask must execute within the same level of the memory hierarchy as its calling task. Additionally, Sequoia tasks have no means of communicating with other tasks executing concurrently on a machine. Although the implementation of `matmul` results in the execution of many parallel tasks, these concurrent tasks do not function as cooperating threads. The lack of shared state among tasks allows parallel tasks to be executed simultaneously using multiple execution units or sequentially on a single processor. Task isolation simplifies parallel programming by obviating the need for synchronization required by cooperating threads. Sequoia language semantics require that output arguments passed to concurrent subtasks do not alias in the calling task's address space. We currently rely on the programmer to ensure this condition holds as this level of code analysis is difficult in the general case.

4.2.3 Task Decomposition

We now introduce Sequoia's *array blocking* and *task mapping* constructs: first-class primitives available to describe portable task decomposition.

In Sequoia, a subset of an array's elements is referred to as an *array block*. For example, `A[0:10]` is the block corresponding to the first 10 elements of the array `A`. The `matmul` task uses the *Range Array Block* blocking function to describe a regular 2D partitioning of its input matrices. In line 16, array blocking syntax is used to divide the matrix `A` into a set of range array blocks each *P* x *Q* in size. Sequoia provides a family of blocking functions via array syntax (see Table 4.1) to facilitate decompositions that range from the simplicity of ranged blocks to the irregularity of arbitrary array gathers.

After defining a blocking for each array, `matmul` iterates over the blocks, recursively calling itself on blocks selected from `A`, `B`, and `C` in each iteration. As introduced in Subsection

4.2.2, the `mappar` construct designates parallel iteration, implying concurrency among subtasks but not asynchronous execution between calling and child tasks. All iterations of a `mappar`, `mapseq`, or `mapreduce` must complete before control returns to the calling task.

Imperative C-style control-flow is permitted in tasks, but use of blocking and mapping primitives is encouraged to facilitate key optimizations performed by the Sequoia compiler and runtime system. A complete listing of Sequoia blocking and mapping constructs is given in Table 4.1.

4.2.4 Task Variants

Figure 4.2 contains two implementations of the `matmul` task, `matmul::inner` and `matmul::leaf`. Each implementation is referred to as a *variant* of the task and is named using the syntax `taskname::variantname`. The variant `matmul::leaf` serves as the base case of the recursive matrix multiplication algorithm. Notice that the Sequoia code to recursively call `matmul` gives no indication of when the base case should be invoked. This decision is made as part of the machine-specific mapping of the algorithm (Section 4.4).

Inner tasks, such as `matmul::inner`, are tasks that call subtasks. Notice that `matmul::inner` does not access elements of its array arguments directly and only passes blocks of the arrays to subtasks. Since a target architecture may not support direct processor access to data at certain hierarchy levels, to ensure code portability, the Sequoia language does not permit inner tasks to directly perform computation on array elements. Instead, inner tasks use Sequoia's mapping and blocking primitives (Section 4.2.3) to structure computation into subtasks. Ultimately, this decomposition yields computations whose working sets fit in leaf memories directly accessible by processing units. An inner task definition is not

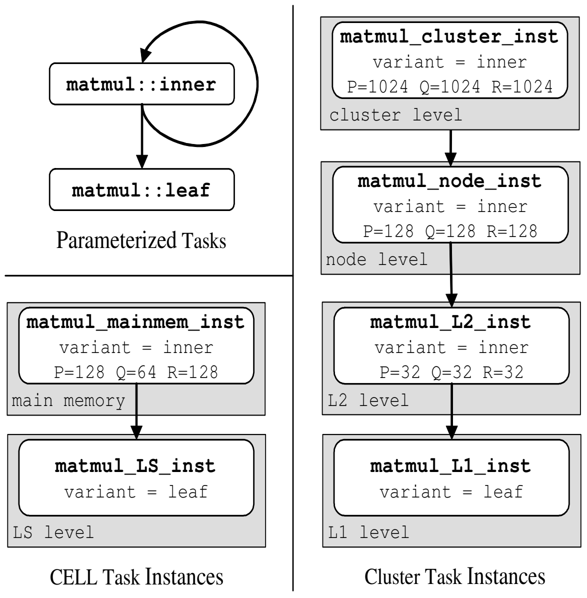

Figure 4.4: The call graph for the parameterized `matmul` task is shown at top left. Specialization to Cell or to our cluster machine generates instances of the task shown at bottom left and at right.

associated with any particular machine memory module; it may execute at any level of the memory hierarchy in which its working set fits.

Leaf tasks, such as `matmul::leaf`, do not call subtasks and operate directly on working sets resident within leaf levels of the memory hierarchy. Direct multiplication of the input matrices is performed by `matmul::leaf`. In practice, Sequoia leaf tasks often wrap platform specific implementations of computational kernels written in traditional languages, such as C or Fortran.

4.2.5 Task Parameterization

Tasks are written in parameterized form to allow for specialization to multiple target machines. *Specialization* is the process of creating *instances* of a task that are customized to operate within, and are mapped to, specific levels of a target machine's memory hierarchy. A task instance defines a variant to execute and an assignment of values to all variant parameters. The Sequoia compiler creates instances for each of the various contexts in which a task is used. For example, to run the `matmul` task on Cell, the Sequoia compiler generates an instance employing the `matmul::inner` variant to decompose large matrices resident in main memory into LS-sized submatrices. A second instance uses `matmul::leaf` to perform the matrix multiplication inside each SPE. On a cluster machine, one `matmul` instance partitions matrices distributed across the cluster into submatrices that fit within individual nodes. Additional instances use `matmul::inner` to decompose these datasets further into L2- and then L1-sized submatrices. While parameterized tasks do not name specific variants when calling subtasks, specialized task instances make direct calls to other instances. The static call graph relating `matmul`'s parameterized task variants is shown at top left in Figure 4.4. Calls among the task instances that result from specialization to Cell and to a cluster are also shown in the figure. Notice that three of the cluster instances, each mapped to a different location of the machine hierarchy, are created from the `matmul::inner` variant (each instance features different argument sizes and parameter values).

Task variants utilize two types of numeric parameters, *array size* parameters and *tunable* parameters. Array size parameters, such as `M`, `N`, and `P` defined in the `matmul` task variants, represent values dependent upon array argument sizes and may take on different values across calls to the same instance. Tunable parameters, such as the integers `U`, `V`, and `X` declared in `matmul::inner` (lines 7-9 of Figure 4.2), are designated using the `tunable` keyword. Tunable parameters remain unbound in Sequoia source code but are statically assigned values during task specialization. Once assigned, tunable parameters are treated as compile-time constants. The most common use of tunable parameters, as illustrated by the matrix multiplication example, is to specify the size of array blocks passed as arguments to subtasks.

Parameterization allows the decomposition strategy described by a task variant to be applied in a variety of contexts, making the task portable across machines and across levels of the memory hierarchy within a single machine. The use of tunable and array size parameters and the support of multiple task variants is key to decoupling the expression of an algorithm from its mapping to an underlying machine. Tasks provide a framework for defining the application-specific space of decisions that must be made during the process of program tuning. In the following section, we describe the process of tuning and targeting Sequoia applications to a machine.

4.3 Sequoia Compiler

The front-end of our system is an adaptation of the Sequoia compiler [Knight et al., 2007]. The compiler (1) transforms a standard AST representation of input Sequoia programs into a machine-independent intermediate representation (IR) consisting of a dependence graph of bulk operations, (2) performs various generic optimizations on this IR, and (3) generates code targeting the runtime interface described in the next chapter. The runtime interface provides a portable layer of abstraction that enables the compiler's generated code to run on a variety of platforms. The original compiler optimization research specifically targeted for the Cell processor was generalized for our runtime system.

The compiler's generic IR optimizations span three main categories:

- locality optimizations, in which data transfer operations are eliminated from the program at the cost of increasing the lifetimes of their associated data objects in a memory level;

- operation grouping, in which “small”, independent operations are fused into larger operations, thereby reducing the relative overheads of the operations; and
- operation scheduling, in which an ordering of operations is chosen to attempt to simultaneously maximize operation concurrency and minimize the amount of space needed in each memory in the machine.

With the exception of the scheduling algorithms, which operate on the entire program at once, all compiler optimizations are local; they apply to a single operation at a time and affect data in either a single memory level or in a pair of adjacent memory levels. The compiler’s optimizations require two pieces of information about each memory in the target machine’s abstract machine model; its size and a list of its properties, specifically whether the memory has the same namespace as any other memories in the machine model (as happens in the SMP target) and whether its logical namespace is distributed across multiple distinct physical memory modules (as in the cluster target). These specific machine capabilities affect the choice of memory movement optimizations the compiler applies. For example, copy elimination is required on machines with a shared namespace to prevent unneeded transfer overhead. A per-machine configuration file provides this information. Aside from these configuration details, the compiler’s optimizations are oblivious to the underlying mechanisms of the target machine, allowing them to be applied uniformly across a range of different machines and also across a range of distinct memory levels within a single machine.

Although the input programs describe a single logical computation spanning an entire machine, the compiler generates separate code for each memory level and instantiates a separate runtime instance for each pair of adjacent levels. Each runtime is oblivious to the details of any runtimes either above or below it.

```
instance {
 name    = matmul_mainmem_inst
 task    = matmul::inner
 runs_at = main_memory
 calls   = matmul_LS_inst
 tunable   P=128,Q=64,R=128
}

instance {
 name    = matmul_LS_inst
 variant = matmul::leaf
 runs_at = LS_level
}
```

```
instance {
 name    = matmul_cluster_inst
 variant = matmul::inner
 runs_at = cluster_level
 calls   = matmul_node_inst
 tunable   P=1024,Q=1024,R=1024
}
instance {
 name    = matmul_node_inst
 variant = matmul::inner
 runs_at = node_level
 calls   = matmul_L2_inst
 tunable   P=128,Q=128,R=128
}
instance {
 name    = matmul_L2_inst
 task    = matmul::inner
 runs_at = L2_cache_level
 calls   = matmul_L1_inst
 tunable   P=32,Q=32,R=32
}
instance {
 name    = matmul_L1_inst
 task    = matmul::leaf
 runs_at = L1_cache_level
}
```

Figure 4.5: Specification for mapping the `matmul` task to a Cell machine (left) and a cluster machine (right).

4.4 Specialization and Tuning

Tasks are generic algorithms that must be specialized before they can be compiled into executable code. Mapping a hierarchy of tasks onto a hierarchical representation of memory requires the creation of task instances for all machine levels. For each instance, a code variant to run must be selected, target instances for each call site must be chosen, and values for tunable parameters must be provided.

One specialization approach is to rely upon the compiler to automatically generate task instances for a target by means of program analysis or a heuristic search through a pre-defined space of possibilities. In Sequoia, the compiler is not required to perform this transformation. Instead, we give the programmer complete control of the mapping and tuning phases of program development. A unique aspect of Sequoia is the *task mapping specification* that is created by the programmer on a per-machine basis and is maintained separately from the Sequoia source. The left half of Figure 4.5 shows the information required to map `matmul` onto a Cell machine. The tunables have been chosen such that submatrices constructed by the instance `matmul_mainmem_inst` can be stored entirely within a single SPE's LS.

In addition to defining the mapping of a task hierarchy to a machine memory hierarchy, the mapping specification also serves as the location where the programmer provides optimization and tuning directives that are particular to the characteristics of the intended target. A performance-tuned mapping specification for `matmul` execution on a cluster is shown in Figure 4.6. The instance `matmul_cluster_inst` runs at the cluster level of the machine hierarchy, so the distribution of array arguments across the cluster has significant performance implications. The instance definition specifies that task argument matrices be distributed using a 2D block-block decomposition consisting of blocks 1024x1024 in size. The definition also specifies that the transfer of subtask arguments to the individual nodes should be software-pipelined across `mappar` iterations to hide the latency of the transfers. As an additional optimization, `matmul_L2_inst` specifies that the system should copy the second and third arguments passed to `matmul::leaf` into contiguous buffers to ensure stride-1 access in the the leaf task.

Mapping specifications are intended to give the programmer precise control over the mapping of a task hierarchy to a machine while isolating machine-specific optimizations in a single location. Performance is improved as details in the mapping specification are refined. While an intelligent compiler may be capable of automating the creation of parts of a new mapping specification, Sequoia's design empowers the performance-oriented programmer to manage the key aspects of this mapping to achieve maximum performance.

```
instance {
 name    = matmul_cluster_inst
 task    = matmul
 variant = inner
 runs_at = cluster_level
 calls   = matmul_node_inst
 tunable   U=1024, X=1024, V=1024

 A distribution = 2D block-block (blocksize 1024x1024)
 B distribution = 2D block-block (blocksize 1024x1024)
 C distribution = 2D block-block (blocksize 1024x1024)

 mappar loop-partition    = grid 4x4
 mappar software-pipeline = true

}
instance {
 name    = matmul_node_inst
 task    = matmul
 variant = inner
 runs_at = node_level
 calls   = matmul_L2_inst
 tunable   U=128, X=128, V=128
}
instance {
 name    = matmul_L2_inst
 task    = matmul
 variant = inner
 runs_at = L2_cache_level
 calls   = matmul_L1_inst
 tunable   U=32, X=32, V=32

 subtask arg A = copy
 subtask arg B = copy
}
instance {
 name    = matmul_L1_inst
 task    = matmul
 variant = leaf
 runs_at = L1_cache_level
}
```

Figure 4.6: A tuned version of the cluster mapping specification from Figure 4.5. The cluster instance now distributes its working set across the cluster and utilizes software-pipelining to hide communication latency.

4.5 Sequoia System

Figure 4.7 shows how the pieces of the system are all put together. First, the user writes their program using Sequoia. The user's source file is fed into the compiler's front-end. This in turn generates a call graph representing the decomposition of the application. The call graph along with a machine description file are fed into the specialization phase of the compiler. This phase maps the call graph onto the specified machine, performs optimizations, and schedules tasks and data transfers between all levels of the machine. The compiler generates C++ code along with runtime API calls which will be compiled by the vendor supplied C++ compiler into machine code.

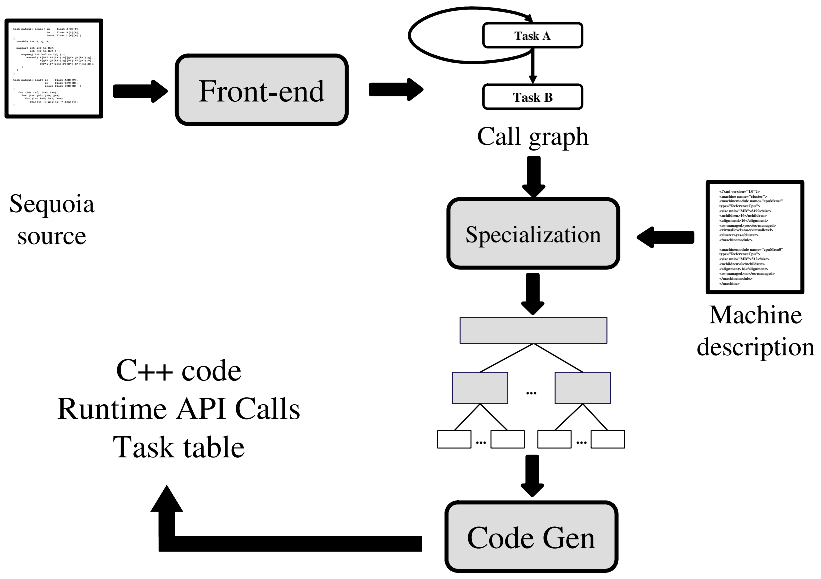

Figure 4.7: Sequoia system overview.

Sequoia Array Blocking Syntax

Range Array Blocks

`A[start0:end0;][start1:end1;][...]`
Generates a blockset containing non-overlapping blocks that tile the multi-dimensional array `A`. Each block is multi-dimensional with size $\texttt{end0} - \texttt{start0} \times \texttt{end1} - \texttt{start1} \times \ldots$.

`A[start0:end0:stride0;][...]`
Generalized form of regular blocking that generates blocksets containing potentially overlapping blocks. The starting array offset, ending offset, and stride between blocks is specified for every dimension of the source array.

Indexed Array Blocks

`A[Idx[i]:Idx[j];Max]`
Generates a set of irregularly-sized blocks from array `A`. Block start and end indices are given by elements in the Idx array. Since the size of the array is dynamic, a maximum block size must be defined for the system to reason about space requirements.

`A[Idx[start0;end0;]]`
Generates a block by gathering elements from the source array `A` using the indices provided `Idx`. The resulting block has all the elements defined by `Idx`. If this syntax is used by a write argument, a scatter operation will occur.

Sequoia Mapping Primitives

`mappar(i=i0 to iM, j=j0 to jN ...)  {...}`
A multi-dimensional for-all loop containing only a subtask call in the loop body. The task is mapped in parallel onto a collection of blocks.

`mapseq(i=i0 to iM, j=j0 to jN ...)  {...}`
A multi-dimensional loop containing only a subtask call in the loop body. The task is mapped in sequentially onto a collection of blocks.

`mapreduce(i=i0 to iM, j=j0 to jN ...)  {...}`
Maps a task onto a collection of blocks, performing a reduction on at least one argument to the task. To support parallel tree reductions, an additional *combiner* subtask is required.

Table 4.1: Sequoia mapping and blocking primitives

Chapter 5

Portable Runtime System

5.1 Runtime Interface

Recall from Section 3.3 the rules of our abstract model. To allow for portability, we need a uniform runtime API that allows us to define the functionality of a tree node. Remember that a tree node consists of a control thread and one memory. A tree node can perform a set of functions against itself and interact with its parent and children using a different set of functions. For a concrete runtime API, we need to define our design requirements:

- Resource allocation: data allocation and naming and initialization of parallel resources.
- Explicit bulk asynchronous communication: transfer lists and transfer commands.
- Parallel execution: launch tasks on children. We need to support asynchronous execution to allow different execution on different subsets of children.

- Synchronization: make sure transfers and tasks complete. There are both asynchronous task and transfer synchronization and sibling synchronization.
- Runtime isolation: runtimes cannot have direct knowledge of each other.

Starting from our abstraction primitive, the tree node, we can divide functionality based on the mode of the tree node. When the node is acting as a parent, it can launch tasks on children and wait for children to complete. When the node is acting as a child, it can perform data communication with its parent and synchronize with siblings. The remaining question is how to handle allocation of resources. We choose to perform resource and data allocation when acting as a parent. We do this for simplicity (e.g. allocation is done in only one mode instead of two) and for practical implementation reasons. We need to initialize parallel resources, which obviously needs to be done via the parent. For data allocation, we could perform allocation on the parent or the child. The problem with doing child centric allocation is how data is allocated in the root memory, especially when dealing with virtual memory levels. If we allocate with the parent, the issue is how data is allocated in the child memory. However, in the later case, the terminal leaf task can define the arrays itself statically or through standard memory allocation. Thus, all intermediate allocation can be handled by the parent mode. Thus, we can define a runtime with two parts, a *top* runtime that defines functionality that a node can access in parent mode, and a *bottom* runtime that defines functionality that a node can access in child mode.

Each runtime straddles the transition between two memory levels. There is only one memory at the parent level, but the child level may have multiple memories; i.e., the memory hierarchy is a tree, where the bottom level memories are children of the top level. An illustration is provided in Figure 5.1.

A runtime in our system provides three main services for code (tasks) running within a memory level: (1) initialization/setup of the machine, including communication resources

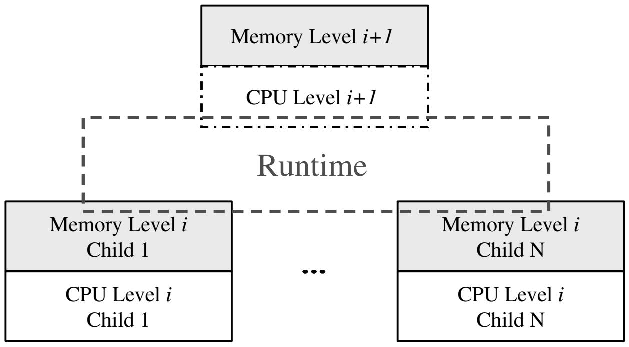

Figure 5.1: A runtime straddles two memory levels.

and resources at all levels where tasks can be executed, (2) data transfers between memory levels using asynchronous bulk transfers between arrays, and (3) task execution at specified (child) levels of the machine.

The interfaces to the top and bottom runtimes have different capabilities and present a different API to clients running at their respective memory levels. A listing of the `C++` public interface of the top and bottom parts of a runtime is given in Figures 5.2 and 5.3. We briefly explain each method in turn.

5.1.1 Top Interface

We begin with Figure 5.2, the API for the top side of the runtime. The top is responsible for both the creation and destruction of runtimes. The constructor requires two arguments: a table of tasks representing the functions that the top level can invoke in the the bottom level of the runtime, and a count of the number of children of the top. At initialization, all runtime resources, including execution resources, are created, and these resources are destroyed at runtime shutdown.

```
// create and free runtimes
Runtime(TaskTable table, int numChildren);
~Runtime();

// allocate and deallocate arrays
Array_t*     AllocArray(Size_t      elmtSize,
                        int         dimensions,
                        Size_t*     dim_sizes,
                        ArrayDesc_t descriptor,
                        int         alignment);
void         FreeArray(Array_t* array);

// register arrays and find/remove arrays using array descriptors
void         AddArray(Array_t array);
Array_t      GetArray(ArrayDesc_t descriptor);
void         RemoveArray(ArrayDesc_t descriptor);

// launch and synchronize on tasks
TaskHandle_t CallChildTask(TaskID_t  taskid,
                           ChildID_t start,
                           ChildID_t end);
void         WaitTask(TaskHandle_t handle);
```

Figure 5.2: The runtime API Top Interface

.

```
// look up array using array descriptor
Array_t      GetArray(ArrayDesc_t descriptor);

// create, free, invoke, and synchronize on transfer lists
XferList*    CreateXferList(Array_t* dst,
                            Array_t* src,
                            Size_t*  dst_idx,
                            Size_t*  src_idx,
                            Size_t*  lengths,
                            int      count);
void         FreeXferList(XferList* list);
XferHandle_t Xfer(XferList* list);
void         WaitXfer(XferHandle_t handle);

// get number of children in bottom level,
// get local processor id, and barrier
int          GetSiblingCount();
int          GetID();
void         Barrier(ChildID_t start, ChildID_t stop);
```

Figure 5.3: The runtime API Bottom Interface.

Our API emphasizes bulk transfer of data between memory levels, and, for this reason, the runtimes directly support arrays. Arrays are allocated and freed via the runtimes (`AllocArray` and `FreeArray`) and are *registered* with the system using the array's reference (`AddArray`) and *unregistered* using the array's descriptor (`RemoveArray`). An array descriptor is a unique identifier supplied by the user when creating the array. Only arrays allocated using the top of the runtime can be registered with the runtime. Registered arrays are visible to the bottom of the runtime via the arrays' descriptors (`GetArray`) and can only be read or written using explicit block transfers.

As mentioned above, tasks are registered with the runtime via a task table when the runtime is created. A request to run a task on multiple children can be performed in a single call to `CallChildTask`. When task f is called, the runtime calling f is passed as an argument to f, thereby allowing f to access the runtime's resources, including registered arrays, transfer functions, and synchronization with other children. Finally, there is a synchronization function `WaitTask` enabling the top of the runtime to wait on the completion of a task executing in the bottom of the runtime.

5.1.2 Bottom Interface

The API for the bottom of the runtime is shown in Figure 5.3. Data is transferred between levels by creating a list of transfers between an array allocated using the top of the runtime and an array at the bottom of the runtime (`CreateXferList`), and requesting that the given transfer list be executed (`Xfer`). Transfers are non-blocking, asynchronous operations, and the client must issue a wait on the transfer to guarantee the transfer has completed (`WaitXfer`). Data transfers are initiated by the children using the bottom of the runtime.

Synchronization is done via a barrier mechanism that can be performed on a subset of the children (`Barrier`). This enables children to synchronize on data and tasks without requiring returning control to the parent. Children can learn their own process id's (`GetID`) and the range of id's of other children (`GetSiblingCount`). These functions were added to enable the children to calculate their portion of data to access. These three API calls were added to allow for greater optimizations and scheduling flexibility by the compiler.

These simple primitives map efficiently to our target machines, providing a mechanism independent abstraction of memory levels. In a multi-level system, the multiple runtimes have no direct knowledge of each other. Traversal of the memory levels, and hence runtimes, is done via task calls. The interface represents, in many respects, the lowest common denominator of many current systems; we explore this further in the presentation of runtime implementations in Section 5.2.

5.2 Runtime Implementations

We implemented our runtime interface for the following platforms: SMP, disk, Cell Broadband Engine, and a cluster of workstations. This section describes key aspects of mapping the interface onto these machines.

5.2.1 SMP

The SMP runtime implements execution on shared-memory machines. A distinguishing feature of shared-memory machines is that explicit communication is not required for correctness, and thus this runtime serves mainly to provide the API's abstraction of parallel execution resources and not the mechanisms to transfer data between memory levels.

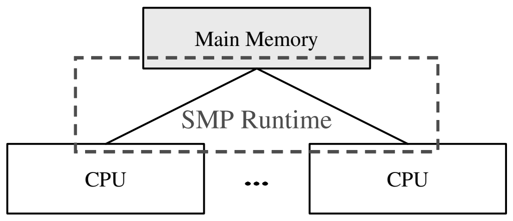

Figure 5.4: Graphical representation of the SMP runtime

On initialization of the SMP runtime a top runtime instance and the specified number of bottom runtimes are created. Each bottom runtime is initialized by creating a POSIX thread, which waits on a task queue for task execution requests. On runtime shutdown, a shutdown request is sent to each child thread; each child cleans up its resources and exits. The top runtime performs a `join` on each of the children's shutdowns, after which the top runtime also cleans up its resources and exits.

`CallChildTask` is implemented by placing a task execution request on the specified child's queue along with a completion notification object. When the child completes the task, it notifies the completion object to inform the parent. When a `WaitTask` is issued on the parent runtime, the parent waits for a task completion signal before returning control to the caller.

Memory is allocated at the top using standard `malloc` routines with alignment specified by the compiler. Arrays are registered with the top of the runtime with `AddArray` and can be looked up via an array descriptor from the bottom runtime instances. Calling `GetArray` from the bottom returns an array object with a pointer to the actual allocated memory from the top of the runtime. Since arrays can be globally accessible, the compiler can opt to

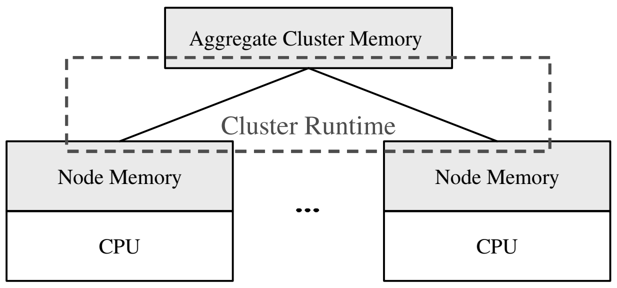

Figure 5.5: Graphical representation of the cluster runtime

directly use this array's data pointers, or issue data transfers by creating `XferLists` with `CreateXferList` and using `Xfer`'s, which are implemented as `memcpy`'s.

5.2.2 Cluster Runtime

The cluster runtime implements execution on distributed memory machines communicating via network interconnects. The aggregate of all node memories is the top (global) level, which is implemented as a distributed shared-memory system, and the individual node memories are the bottom (local) level, with each cluster node as one child of the top level. Similar to the disk, the cluster's aggregate memory space is logically above any processor's local memory, and the runtime API allows the local level to read/write portions of the potentially distributed arrays. We implement the cluster runtime with a combination of Pthreads and MPI-2 [MPIF, 1996].

On initialization of the runtime, node 0 is designated to execute the top level runtime functions. All nodes initialize as bottom runtimes and wait for instructions from node 0. Two threads are launched on every node: an *execution thread* to handle the runtime calls and the execution of specified tasks, and a *communication thread* to handle data transfers, synchronization, and task call requests across the cluster.

Bottom runtime requests are serviced by the execution thread, which identifies and dispatches data transfer requests to the communication thread, which performs all MPI calls. Centralizing all asynchronous transfers in the communication thread simplifies implementation of the execution thread and works around issues with multi-threading support in several MPI implementations.

We provide a distributed shared-memory (DSM) implementation to manage memory across the cluster. However, unlike conventional DSM implementations, we need not support fully general memory or coherence. All access to memory from the bottom of the runtime must be explicit and in bulk, and the parallel memory hierarchy programming model forbids aliasing. The strict access rules on arrays give us great flexibility in strategies for allocating arrays across the cluster. We use an *interval tree* [Cormen et al., 2001] per allocated array, which allows specifying a distribution on a per array basis. Because of the copy-in, copy-out semantics of access to arrays passed to tasks in the Sequoia programming model, we can support complex data replication where distributions partially overlap. Unlike traditional DSM implementations where data consistency and coherence must be maintained by the DSM layer, the programming model asserts this property directly. For the purposes of this dissertation, we use only simple block-cyclic data distributions as complex distributions are not currently generated by the compiler.

We use MPI-2 single-sided communication to issue `gets` and `puts` on remote memory systems. If the memory region requested is local to the requesting node and the requested memory region is contiguous, we can directly use the memory from the DSM layer by simply updating the destination pointer, therefor reducing memory traffic. However, the response of a data transfer in this case is not instantaneous since there is communication between the execution and communication threads as well as logic to check for this condition. If the data is not contiguous in memory on the local node, we must use `memcpys` to construct a contiguous block of the requested data.

When the top of the runtime (node 0) launches a task execution on a remote node, node 0's execution thread places a task call request on its command queue. The communication thread monitors the command queue and sends the request to the specified node. The target node's communication thread receives the request and adds the request to the task queue, where it is subsequently picked up and run by the remote node's execution thread. Similarly, to perform synchronization an execution thread places a barrier request in the command queue and waits for a completion signal from the communication thread.

To implement a barrier, the communication thread sends a barrier request to the specified node set and then monitors barrier calls from other nodes' communication threads. Once all nodes have sent a barrier message for the given barrier, the communication thread notifies the execution thread, which returns control to the running task. One interesting note is that during a barrier the communication thread must continue to act on requests for barriers other than the barrier being waited on, as the compiler may generate barriers for different subsets of nodes.

5.2.3 Cell

The Cell Broadband Engine comprises a PowerPC (PPE) core and eight SPEs. At initialization, the top of the runtime is created on the PPE and an instance of the bottom of the runtime is started on each of the SPEs. We use the IBM Cell SDK 2.1 and `libspe2` for command and control of SPE resources [IBM, 2007b].

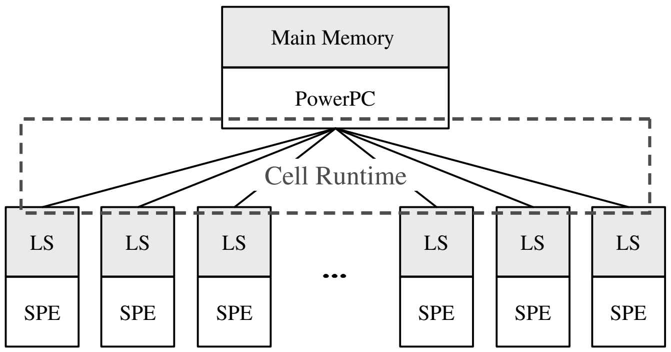

Figure 5.6: Graphical representation of the Cell runtime

Each SPE waits for commands to execute tasks via mailbox messages. For the PPE to launch a task in a given SPE, it signals that SPE's mailbox and the SPE loads the corresponding code overlay of the task and begins execution—SPE's have no instruction cache and so code generated for the SPE must include explicit code overlays to be managed by the runtime. Note that being able to move code through the system and support code overlays is one of the reasons a task table is passed to the runtime at initialization.

The majority of the runtime interfaces for data transfer have a direct correspondence to functions in the Cell SDK. Creating a `XferList` maps to the construction of a DMA list for the `mfc_getl` and `mfc_putl` SDK functions which are executed on a call to `Xfer`. `XferWait` waits on the tag used to issue the DMA. Allocation in a SPE is mapped to offsets in a static array created by the compiler, guaranteeing the DMA requirement of 16 byte memory alignment. Synchronization between SPEs is performed through mailbox signaling routines.

The PPE allocates memory via `posix_memalign` to align arrays to the required DMA transfer alignment. To run a task in each SPE, the PPE sends a message with a task ID corresponding to the address of the task to load as an overlay. Overlays are created for each leaf

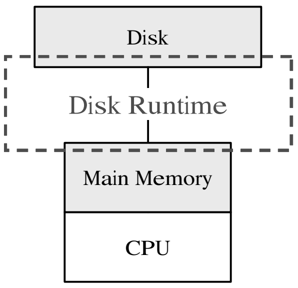

Figure 5.7: Graphical representation of the Disk runtime

task by the build process provided by the compiler and are registered with the runtime on runtime initialization.

5.2.4 Disk

The disk runtime is interesting because the disk's address space is logically above the main processor's. Specifically, the disk is the top of the runtime and the processor is the bottom of the runtime, which can pull data from and push data to the parent's (disk's) address space. Our runtime API allows a program to read/write portions of arrays from its address space to files on disk. Arrays are allocated at the top using `mkstemp` to create a file handle in temporary space. This file handle is mapped to the array descriptor for future reference. Memory is actually allocated by issuing a `lseek` to the end of the file, using the requested size as the seek value, and a sentinel is written to the file to verify that the memory could be allocated on disk.

Data transfers to and from the disk are performed with the Linux Asynchronous I/O API. The creation of a transfer list (`XferList` in Figure 5.3) constructs a list of `aio_cb` structures suitable for a transfer call using `lio_listio`. Memory is transferred using `lio_listio`

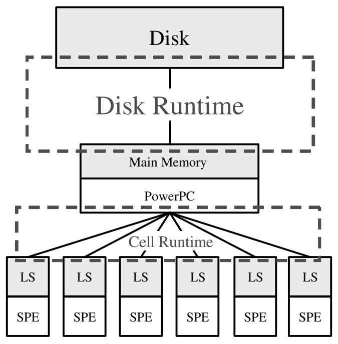

Figure 5.8: Hierarchical representation of the composed Disk and PS3 runtimes

with the appropriate `aio_read` or `aio_write` calls. On a `WaitXfer`, the runtime checks the return status of each request and issues an `aio_suspend` to yield the processor until the request completes.

`CallChildTask` causes the top to execute the function pointer and transfer control to the task. The disk itself has no computational resources, and so the disk level must always be the root of the memory hierarchy—it can never be a child where leaf tasks can be executed.

5.3 Multi-Level Machines With Composed Runtimes

Since the runtimes share a generic interface and have no direct knowledge of each other, the compiler can generate code that initializes a runtime per pair of adjacent memory levels in the machine. Which runtimes to select is machine dependent and is given by the programmer in a separate specification of the machine architecture; the actual "plugging together" of the runtimes is handled by the compiler as part of code generation.

Two key issues are how isolated runtimes can be initialized at multiple levels and how communication can be overlapped with computation. In our system, both of these are

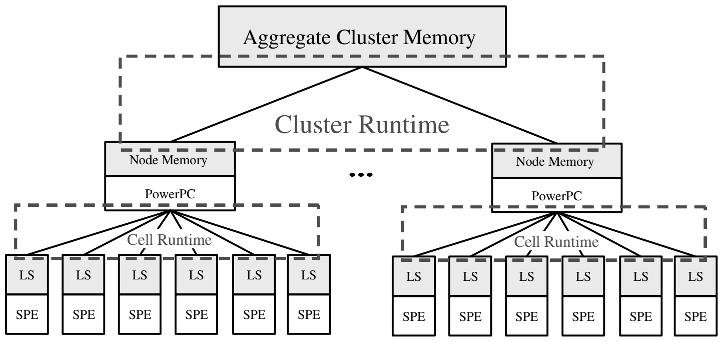

Figure 5.9: Hierarchical representation of the composed Cluster and PS3 runtimes

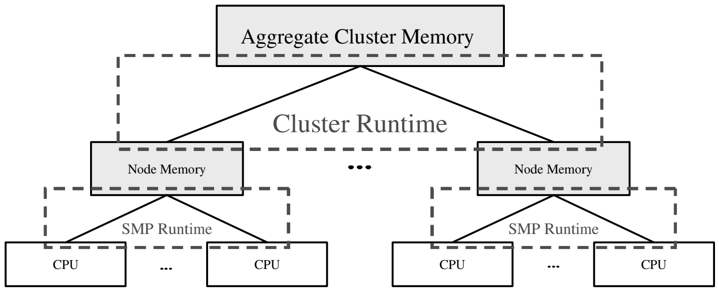

Figure 5.10: Hierarchical representation of the composed Cluster and SMP runtimes

handled by appropriate runtime API calls generated by the compiler. Initializing multiple runtimes is done by initializing the topmost runtime, then calling a task on all children that initializes the next runtime level, and so on, until all runtimes are initialized. Shutdown is handled similarly, with each runtime calling a task to shutdown any child runtimes, waiting, and then shutting down itself. To overlap communication and computation, the compiler generates code that initiates a data transfer at a parent level and requests task execution on child levels. Thus, a level in the memory hierarchy can be fetching data while lower levels can be performing computation.

For this dissertation, we have chosen several system configurations to demonstrate composition of runtimes. Currently available Cell machines have a limited amount of memory, 512 MB per Cell on the IBM blades and 256 MB of memory on the Sony Playstation 3, which uses a Cell processor with 6 SPEs available when running Linux. Given the high performance of the processor, it is common to have problem sizes limited by available memory. With the programming model, compiler, and runtimes presented here, we can compose the Cell runtime and disk runtime to allow running out of core applications on the Playstation 3 without modification to the user's Sequoia code. We can compose the cluster and Cell runtimes to leverage the higher throughput and aggregate memory of a cluster of Playstation 3's. Another common configuration is a cluster of SMPs. Instead of requiring the programmer to write MPI and Pthreads/OpenMP code, the programmer uses the cluster and SMP runtimes to run Sequoia code unmodified.

Chapter 6

Evaluation

We evaluate our system using several applications written in Sequoia (Table 6.1). We show that using our runtime system, we can run unmodified Sequoia applications on a variety of two-level systems (Section 6.3) as well as several multi-level configurations (Section 6.2) with no source level modifications, only remapping and recompilation. Our evaluation centers on how efficiently we can utilize each configuration's bandwidth and compute resources as well as the overheads incurred by our abstraction. We also compare the performance of the applications running on our system against other best known implementations. Despite our uniform abstraction, we maximize bandwidth or compute resources for most applications across our configurations and offer competitive performance against other implementations.

SAXPY	BLAS L1 saxpy
SGEMV	BLAS L2 sgemv
SGEMM	BLAS L3 sgemm
CONV2D	Convolution using a 9x9 filter with a large single-precision floating point input signal obeying non-periodic boundary conditions.
FFT3D	Discrete Fourier transform of a single-precision complex N^3 dataset. Complex data is stored in struct-of-arrays format.
GRAVITY	An $O(N^2)$ N-body stellar dynamics simulation on 8192 particles for 100 time steps. We operate in single-precision using Verlet update and the force calculation is acceleration without jerk [Fukushige et al., 2005].
HMMER	Fuzzy protein string matching using Hidden Markov Model evaluation. The Sequoia implementation of this algorithm is derived from the formulation of HMMER-search for graphics processors given in [Horn et al., 2005] and is run on a fraction of the NCBI non-redundant database.

Table 6.1: Our application suite

6.1 Two-level Portability

For the two-level portability tests, we utilize the following concrete machine configurations:

- The **SMP** runtime is mapped to an 8-way, 2.66 GHz Intel Pentium4 Xeon machine with four dual-core processors and 8 GB of memory.
- The **cluster** runtime drives a cluster of 16 nodes, each with dual 2.4 GHz Intel Xeon processors, 1 GB of memory, connected with Infiniband 4X SDR PCI-X HCAs. With MVAPICH2 0.9.8 [Huang et al., 2006] using VAPI, we achieve ∼400 MB/s node to node.[1] We utilize only one processor per node for this two-level configuration for direct comparison to previous work.
- The **Cell** runtime is run both on a single 3.2 GHz Cell processor with 8 SPEs and 1 GB of XDR memory in an IBM QS20 bladeserver [IBM, 2007a], as well as on the

[1]MVAPICH2 currently exhibits a data integrity issue on our configuration limiting maximum message length to <16KB resulting in a 25% performance reduction over large transfers using MPI-1 calls in MVAPICH

	SMP	Disk	Cluster	Cell	PS3	Cluster of SMPs	Disk + PS3	Cluster of PS3s
SAXPY	16M	384M	16M	16M	16M	16M	64M	16M
SGEMV	8Kx4K	16Kx16K	8Kx4K	8Kx4K	8Kx4K	8Kx4K	8Kx8K	8Kx4K
SGEMM	4Kx4K	16Kx16K	4Kx4K	4Kx4K	4Kx2K	8Kx8K	8Kx8K	4Kx4K
CONV2D	8Kx4K	16Kx16K	8Kx4K	8Kx4K	4Kx4K	8Kx4K	8Kx8K	8Kx4K
FFT3D	256^3	512^3	256^3	256^3	128^3	256^3	256^3	256^3
GRAVITY	8192	8192	8192	8192	8192	8192	8192	8192
HMMER	500 MB	500 MB	500 MB	500 MB	160 MB	500 MB	320 MB	500 MB

Table 6.2: Dataset sizes used for each application for each configuration

3.2 GHz Sony Playstation 3 (PS3) Cell processor with 6 SPEs and 256 MB of XDR memory [Sony, 2007].

- The **disk** runtime is run on a 2.4 GHz Intel Pentium4 with an Intel 945P chipset, a Hitachi 180GXP 7,200 RPM ATA/100 hard drive, and 1 GB of memory.

Application performance in effective GFLOPS is shown in Table 6.3. Information about the dataset sizes used for each configuration are provided in Table 6.2. The time spent in task execution, waiting on data transfer, and runtime overhead is shown in Figures 6.1-6.5. We also show the time spent in kernel execution as percentage of the total execution and the percentage of peak bandwidth achieved in Figures 6.6-6.10. In order to provide a baseline performance metric to show the tuning level of our kernels, we provide results from a 2.4 GHz Intel Pentium4 Xeon machine with 1 GB of memory directly calling our computation kernel implementations in Table 6.3. Our application kernels utilize the fastest implementations publicly available. For configurations using x86 processors, we use FFTW [Frigo, 1999] and the Intel MKL[Intel, 2005], and for configurations using one or more Cell processors, we use the IBM SPE matrix [IBM, 2007b] library. All other leaf tasks are our own best effort implementations, hand-coded in SSE or Cell SPE intrinsics.

Several tests, notably SAXPY and SGEMV, are limited by memory system performance on all platforms but have high utilization of bandwidth resources. SAXPY is a pure streaming bandwidth test and achieves $\sim$40 MB/s from our disk runtime, 3.7 GB/s from our SMP machine, 19 GB/s from the Cell blade, and 17 GB/s on the PS3, all of which are very close

	Baseline	SMP	Disk	Cluster	Cell	PS3
SAXPY	0.3	0.7	0.007	4.9	3.5	3.1
SGEMV	1.1	1.7	0.04	12	12	10
SGEMM	6.9	45	5.5	90	119	94
CONV2D	1.9	7.8	0.6	24	85	62
FFT3D	0.7	3.9	0.05	5.5	54	31
GRAVITY	4.8	40	3.7	68	97	71
HMMER	0.9	11	0.9	12	12	7.1

Table 6.3: Two-level Portability - Application performance (GFLOPS) on a 2.4 GHz P4 Xeon (Baseline), 8-way 2.6 6GHz Xeons (SMP), with arrays on a single parallel ATA drive (Disk), a cluster of 16 2.4 GHz P4 Xeons connected with Infiniband (Cluster), a 3.2 GHz Cell processor with 8 SPEs (Cell), and a Sony Playstation 3 with a 3.2 GHz Cell processor and 6 available SPEs.

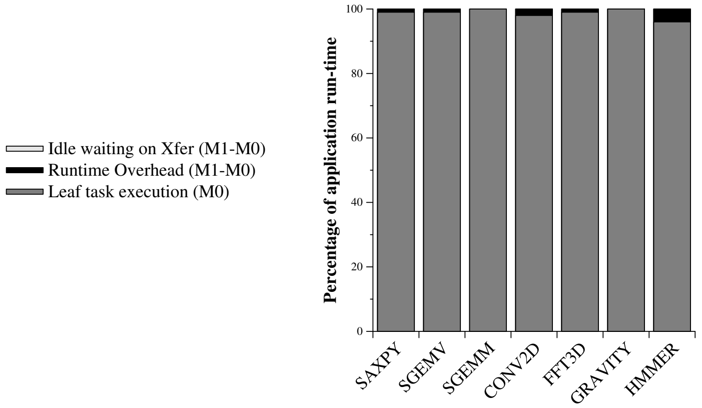

Figure 6.1: Execution time breakdown for each benchmark when running on the SMP runtime

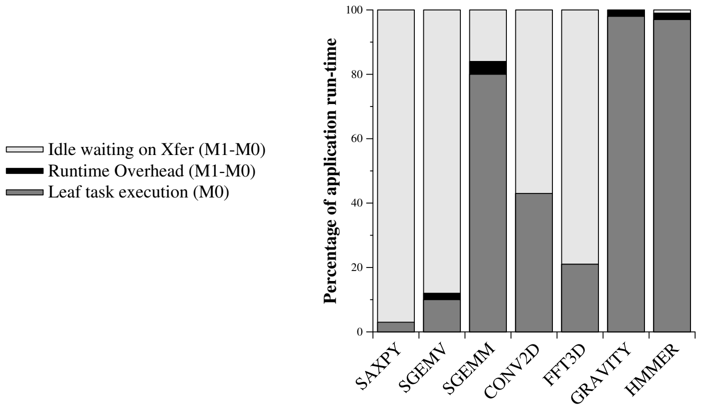

Figure 6.2: Execution time breakdown for each benchmark when running on the Disk runtime

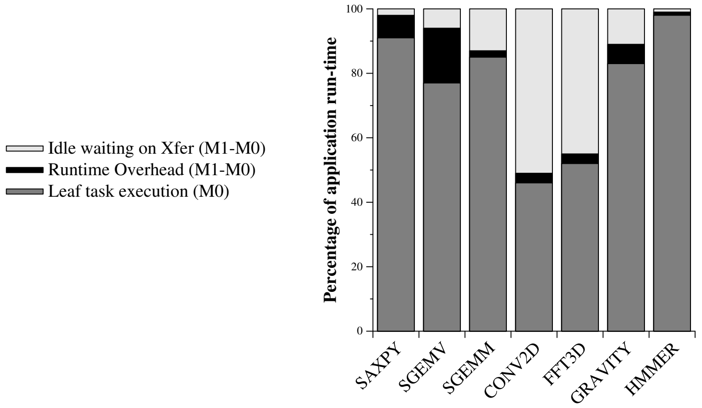

Figure 6.3: Execution time breakdown for each benchmark when running on the Cluster runtime

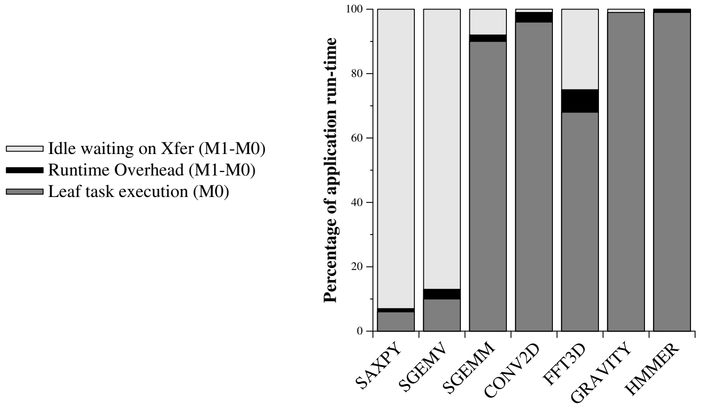

Figure 6.4: Execution time breakdown for each benchmark when running with the Cell runtime on the IBM QS20 (single Cell)

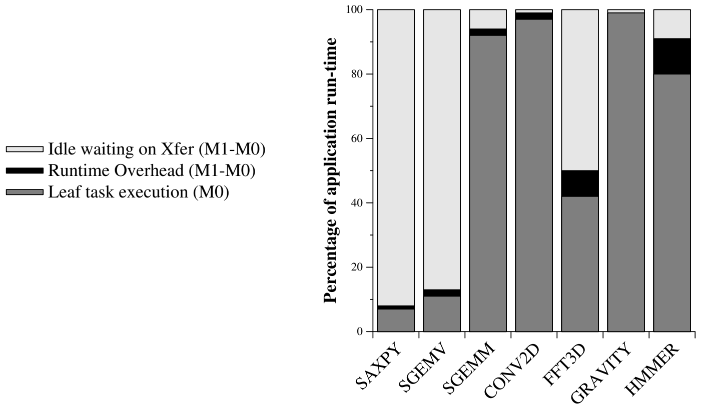

Figure 6.5: Execution time breakdown for each benchmark when running with the Cell runtime on the Sony Playstation 3

to peak available bandwidth on these machines. The cluster provides an amplification effect on bandwidth since there is no inter-node communication required for SAXPY, and we achieve 27.3 GB/s aggregate across the cluster. SGEMV performance behaves similarly, but compiler optimizations result in the x and y vectors being maintained at the level of the processor and, as a result, less time is spent in overhead for data transfers. Since `Xfers` are implicit in the SMP runtime, it has no direct measurement of memory transfer time, and shows no idle time waiting on `Xfers` in Figure 6.1. However, these applications are limited by memory system performance as can be seen in the bandwidth utilization graphs in Figures 6.6-6.10.

FFT3D has complex access patterns. On Cell, we use a heavily optimized 3-transpose version of the code similar to the implementation of Knight et al. [Knight et al., 2007]. On the Cell blade, we run a 256^3 FFT, and our performance is competitive with the large FFT implementation for Cell from IBM [Chow et al., 2005], as well as the 3D FFT implementation of Knight et al. [Knight et al., 2007]. On the PS3, 128^3 is the largest cubic 3D FFT we can fit in-core with the 3-transpose implementation. With this smaller size, the cost of a DMA, and therefore the time waiting on DMAs, increases. Our other implementations, running on machines with x86 processors, utilize FFTW for a 2D FFT on XY planes followed by a 1D FFT in Z to compute the 3D FFT. On the SMP system, we perform a 256^3 FFT and get a memory system limited speedup of 4.7 on eight processors. As can be seen from Figure 6.6, we achieve below peak memory bandwidth because of the memory access pattern of FFT3D. We perform a 512^3 FFT from disk, first bringing XY planes in-core and performing XY 2D FFTs, followed by bringing XZ planes in-core and performing multiple 1D FFTs in Z. Despite reading the largest possible blocks of data at a time from disk, we are bound by disk access performance, with most of the time waiting on memory transfers occurring during the Z-direction FFTs. This read pattern causes us to achieve only $\sim$40% of the peak disk streaming bandwidth. For the cluster runtime, we distribute the XY

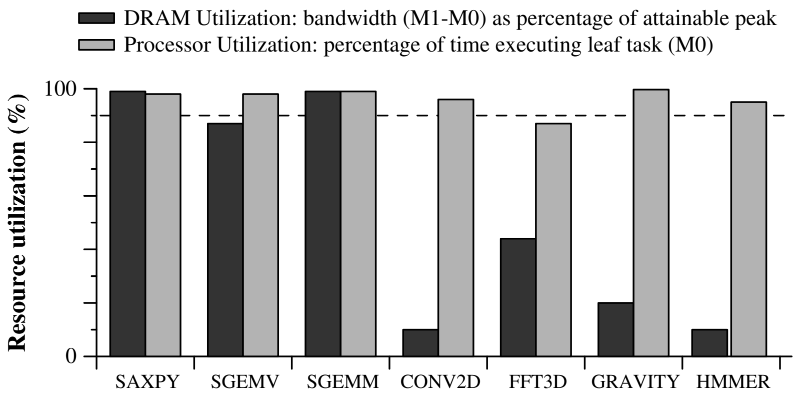

Figure 6.6: Resource utilization on smp

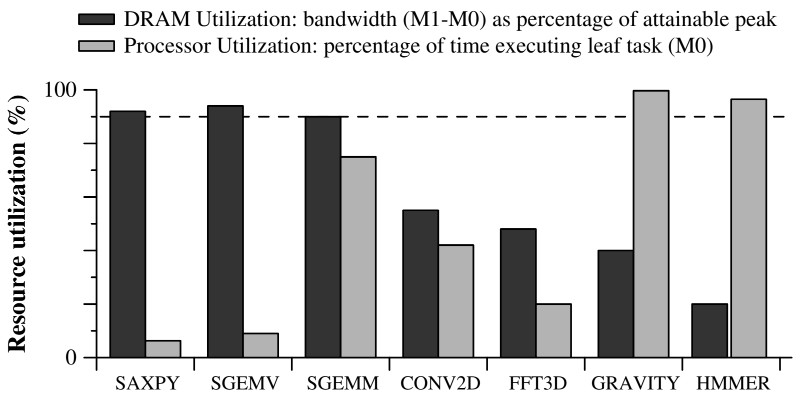

Figure 6.7: Resource utilization on disk

planes across the cluster, making XY 2D FFTs very fast. However, the FFTs in Z become expensive, and we become limited by the cluster interconnect performance.

CONV2D with a 9x9 window tends to be bound by memory system performance on several of our platforms. From disk, we once again achieve very close to the maximum read performance available. On the cluster, we distribute the arrays across nodes and, thus, have to read parts of the image from neighboring nodes and become limited by the interconnect performance. For the Cell platforms, we are largely compute limited. However, since we use software-pipelined transfers to the SPEs generated by the compiler to hide memory

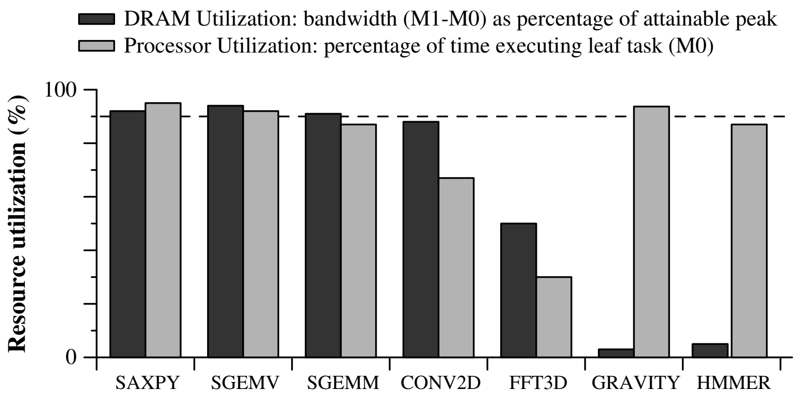

Figure 6.8: Resource utilization on cluster

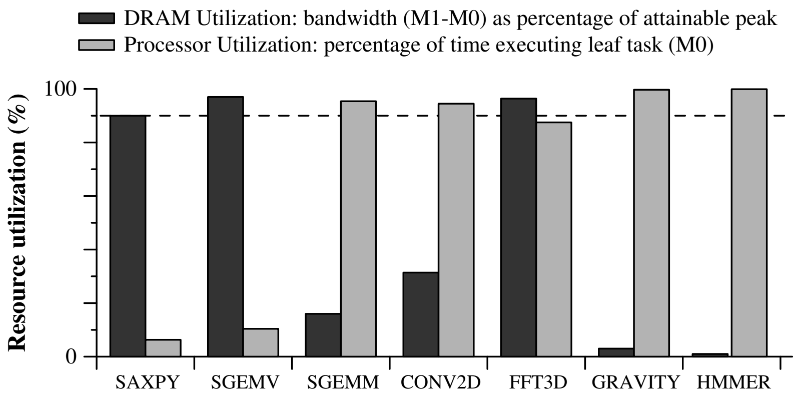

Figure 6.9: Resource utilization on Cell

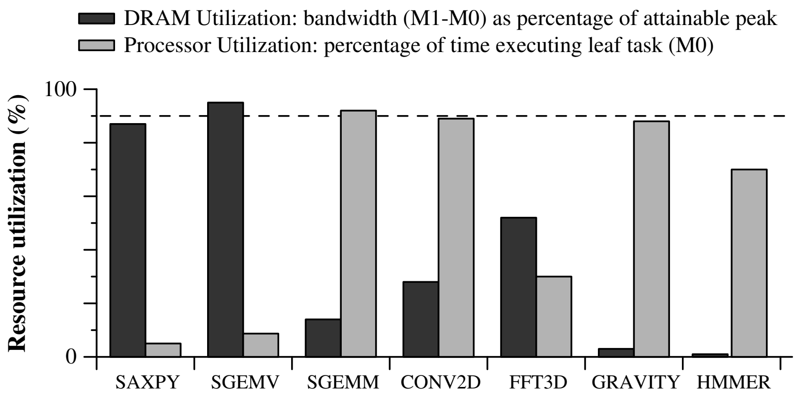

Figure 6.10: Resource utilization on Sony Playstation 3

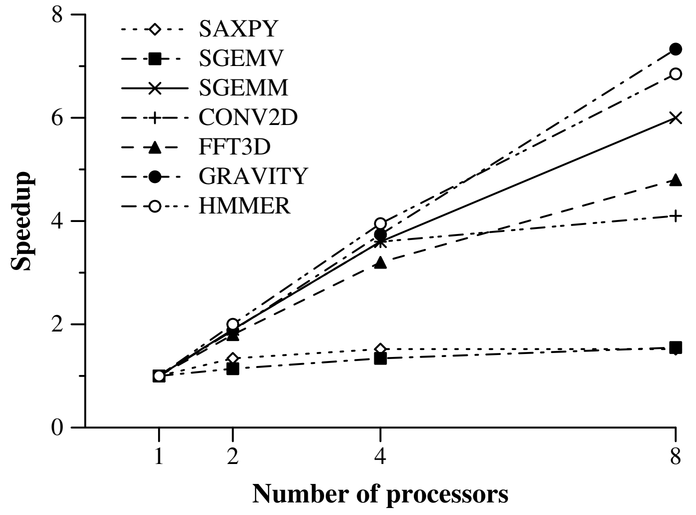

Figure 6.11: SMP application scaling

latency, leading to smaller array blocks on which to compute, the overhead of the support set for the convolution begins to become large and limits our performance.

SGEMM is sufficiently compute intensive that all platforms start to become bound by task execution instead of memory system performance. On our 8-way SMP machine, we achieve a speedup of 6 and observe 3.8 GB/s from the memory system, which is close to peak memory performance. Our performance from disk for a 16K by 16K matrix multiply is similar in performance to the in-core performance of a 4K by 4K matrix used for our baseline results. Our cluster performance for a distributed 4K by 4K matrix multiply achieves a speedup of 13. On Cell, we are largely computation bound, and the performance scales with the number of SPEs as can be seen from the performance on the IBM blade vs. the PS3.

HMMER and GRAVITY are compute bound on all platforms. The only noticeable time spent in anything but compute for these applications is on the cluster runtime where GRAVITY is idle waiting for memory transfers caused by fetching updated particle locations each

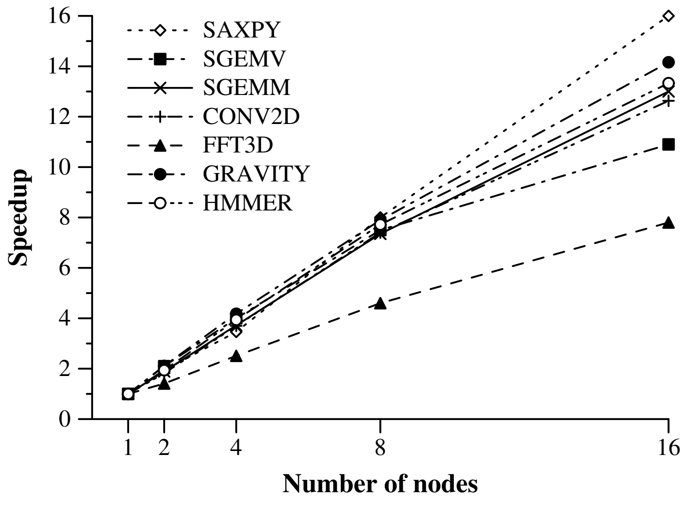

Figure 6.12: Cluster application scaling

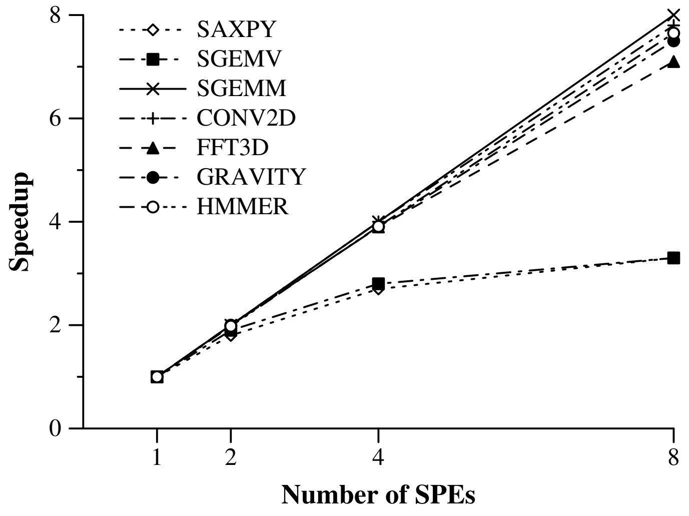

Figure 6.13: Cell application scaling

time-step, and on the PS3 with HMMER where we can only fit 160 MB of the NCBI non-redundant database in memory (the sequences were chosen at random). All other platforms run the same database subset used in Fatahalian et al. [Fatahalian et al., 2006] for results parity, which, including the surrounding data structures, totals more than 500 MB. For disk, we do not bring the database in-core. Instead, we load the database as needed from disk; yet, performance closely matches that of the in-core version. The SMP exhibits super-linear scaling because these processors have larger L2 caches (1 MB vs. 512 KB) than our baseline machine. The cluster achieves a speedup of 13 on 16 nodes, or 83% of the maximum achievable speedup, with much of the difference due to load imbalance between nodes when processing different length sequences.

In general, for these applications and dataset sizes, the overhead of the runtime implementations is low. The disk and cluster runtimes are the most expensive. The disk runtime's overhead is due to the kernel calls required for asynchronous I/O. The cluster runtime's overhead is due to the DSM emulation and threading API calls. The overheads are measured as all critical path execution time other than waiting for memory transfers and leaf task execution. Thus our overhead numbers account for runtime logic, including transfer list creation and task calling/distribution, and time in barriers. The time spent issuing memory transfers is included within the transfer wait times.

The consequences of our implementation decisions for our Cell and cluster runtimes can be seen in the performance differences between our system and the custom Cell backend from Knight et al. [Knight et al., 2007] and the high-level cluster runtime from Fatahalian et al. [Fatahalian et al., 2006]. When scaling the performance results from Knight et al. to account for clock rate differences between their 2.4 GHz Cell processor and our 3.2 GHz Cell processor, we see that our runtime incurs slightly more overhead than their system. For example, for SGEMM, scaling the previously reported numbers, they would achieve 128 GFLOPS, whereas we achieve 119 GFLOPS, a difference of 7%. For FFT, GRAVITY, and HMMER, our performance is 10%, 13%, and 10% lower, respectively, than previously reported results. This overhead is the difference between our more general runtime and their custom build environment which produces smaller code, thus allowing for slightly larger resident working sets in the SPE, more optimization by the compiler by emitting static bounds on loops, and other similar assistance for the IBM compiler tool-chain to heavily optimize the generated code.

The differences between our cluster runtime implementation and that of Fatahalian et al. [Fatahalian et al., 2006] is in their implementation, much of the work performed dynamically is now performed at compiler time. Since we have a much thinner layer, we have less runtime logic overhead in general, and, for some applications, we achieve better performance as the generated code has static loop bounds and alignment hints. SAXPY, SGEMV, and GRAVITY are faster than the previous cluster runtime implementation mainly due to to these improvements. FFT3D performance is lower on our implementation as compared to their implementation due to the lower achievable bandwidth when using MPI-2 single-sided communication through MVAPICH2, as noted above.

We can also explore the performance of our applications running in Sequoia against several best-known implementations. The Intel MKL provides support for execution of SGEMM on a cluster of workstations as well as an SMP system. Using the same system configurations and dataset sizes, Intel Cluster MKL achieves 101 GFLOPS and on SMP Intel MKL achieves 45 GFLOPS compared to our performance of 91 GFLOPS and 44 GFLOPS respectively. We are within 10% of the Intel Cluster MKL performance and within 3% of the SMP MKL performance. For FFT3D, FFTW 3.2 alpha 2 provides an optimized implementation for a cluster of workstations and SMP machines, as well as an experimental version for Cell. For the cluster and SMP, FFTW achieves 5.3 GFLOPS and 4.2 GFLOPS respectively vs. 5.5 GFLOPS and 3.9 GFLOPS from our Sequoia implementations. We actually outperform the cluster version of FFTW and are less than 10% slower than the SMP version. In the case of SMP, we implement the 3D FFT as stated above as a 2D combined with a 1D pass. When calling FFTW directly, we ask the library to perform a 3D FFT and let FFTW search for the best implementation.

Although there are not matching implementations for HMMer and Gravity for our platforms, we can compare performance against best known implementations tuned for specialized architectures. ClawHmmer [Horn et al., 2005] achieves 9.4 GFLOPS on an ATI X1900XT. On Cell and SMP we surpass this performance, achieving 12 GFLOPS and 11 GFLOPS respectively. In the case of Gravity, we can compare our performance against an implementation running on a custom accelerator designed specifically for this type of calculation, the Grape-6A [Fukushige et al., 2005], which achieves 2 billion interactions/s. Our Cell and PS3 implementations achieve 4 billion interactions/s and 3 billion interactions/s, surpassing the performance of the specialized hardware. This demonstrates that our Sequoia implementations running on our runtime system are very efficient at using available machine resources.

6.2 Multi-level Portability

We compose runtimes to explore multi-level portability. By composing the cluster and SMP runtimes, we can execute on a cluster of SMP nodes comprised of four 4-way Intel 3.16 GHz Pentium4 Xeon machines connected with GigE; we utilize two out of the four processors in the node for our tests. Using MPICH2 [ANL, 2007], we achieve $\sim$80 MB/s node-to-node for large transfers. By composing the disk and Cell runtimes, we can overcome the memory limitations of the PS3 to run larger, out-of-core datasets from the 60 GB disk in the console. Further, we can combine the cluster and Cell runtimes to drive

	Cluster of SMPs	Disk + PS3	Cluster of PS3s
SAXPY	1.9	0.004	5.3
SGEMV	4.4	0.014	15
SGEMM	48	3.7	30
CONV2D	4.8	0.48	19
FFT3D	1.1	0.05	0.36
GRAVITY	50	66	119
HMMER	14	8.3	13

Table 6.4: Multi-level Portability - Application performance (GFLOPS) on four 2-way, 3.16 GHz Intel Pentium 4 Xeons connected via GigE (Cluster of SMPs), a Sony Playstation 3 bringing data from disk (Disk + PS3), and two PS3's connected via GigE (Cluster of PS3s).

two PS3's connected via GigE, achieving a higher peak FLOP rate and support for larger datasets.

The raw GFLOPS rates for our applications are shown in Table 6.4. Figures 6.14-6.16 show a breakdown of the total execution time for each application on each configuration, including the task execution time in the lowest level (M0), the overhead between the bottom two levels (M1-M0), the time idle waiting on `Xfer`'s between the bottom levels (M1-M0), overhead between the top two memory levels (M2-M1), and time idle waiting on `Xfer`'s between the top levels (M2-M1). Memory system performance of the slowest memory system dominates the memory limited applications, whereas the compute limited applications are dominated by execution time in the bottom-most memory level. On all three configurations, SAXPY, SGEMV, CONV2D, and FFT3D become bound by the performance of the memory system, while GRAVITY and HMMER, which are very math intensive, are compute bound.

For SAXPY and SGEMV on the cluster of SMPs, we get a bandwidth amplification effect similar to the cluster runtime from above. Since the data is local to the node, there are no actual memory transfers, only the overhead of the runtime performing this optimization. SAXPY and SGEMV also exhibit a larger overhead for M1-M0 which can be attributed to larger scheduling differences and differing start and completion times of the executing

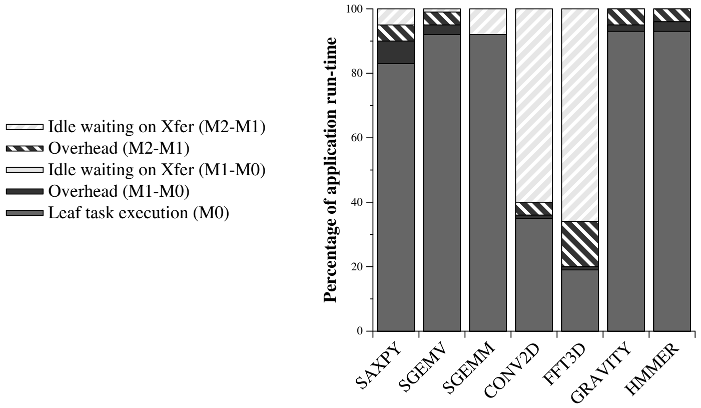

Figure 6.14: Execution time breakdown for each benchmark when running on a Cluster of SMPs

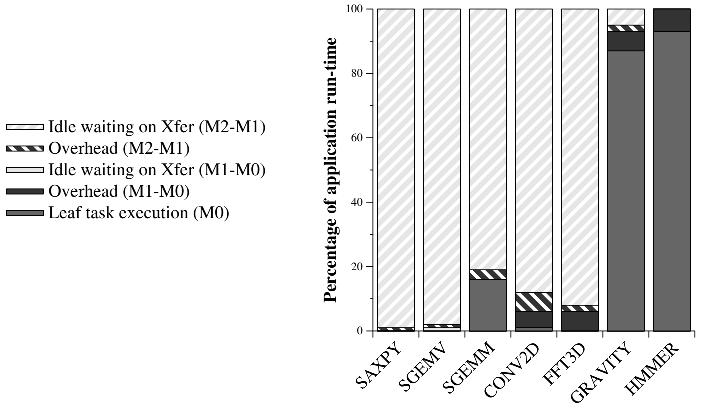

Figure 6.15: Execution time breakdown for each benchmark when running on a Disk+Sony Playstation 3

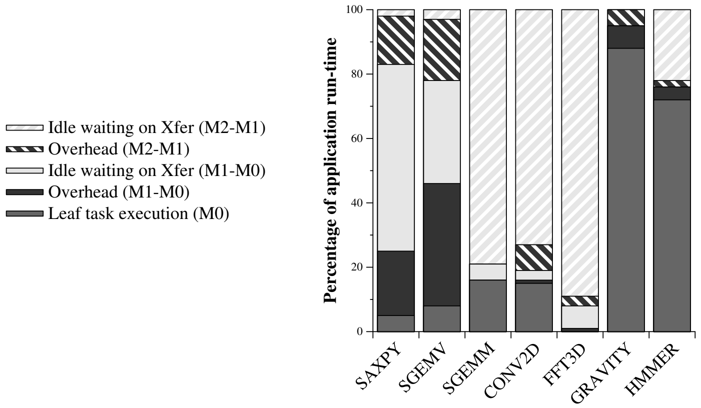

Figure 6.16: Execution time breakdown for each benchmark when running on a Cluster of Sony Playstation 3's

tasks. CONV2D has the scaling behavior of the 8-way SMP from Section 6.3 but with the bandwidth limitations of the GigE interconnect for transmission of support regions. FFT3D becomes bound by the interconnect performance during the FFTs in the Z direction, similar to the cluster results from Section 6.3. SGEMM using 8K by 8K matrices is compute bound but we are able to hide most of the data transfer time. HMMER and GRAVITY are insensitive to the memory performance of this configuration and scale comparably to the 8-way SMP system when clock rate differences are taken into account.

By composing the disk and Cell runtimes, we are able to overcome the memory size limitations on the PS3 and handle larger datasets. However, attaching a high performance processor to a 30 MB/s memory system has a large impact on application performance if the compute to bandwidth ratio is not extremely high. Only HMMER and GRAVITY achieve performance close to the in-core versions, with performance limited mainly by overheads in the runtimes. We ran HMMER with a 500 MB portion of the NCBI non-redundant database from disk. As with the disk configuration from Section 6.3 for GRAVITY, at each timestep, the particles are read from and written to disk. For SGEMM, there is not enough main memory currently available to allow us to work on large enough blocks in-core to hide the transfer latency from disk and we currently spend 80% of our time waiting on disk. All the other applications perform at the speed of the disk, but we are able to run much larger instances than possible in-core.

We are also able to drive two PS3's connected with GigE by combining the cluster and Cell runtimes. HMMER and GRAVITY nearly double in performance across two PS3's as compared to the performance of a single PS3, and HMMER can run on a database twice the size. The combined runtime overhead on GRAVITY is $\sim$8% of the total execution time. For HMMER, we spend $\sim$15% of the execution time waiting on memory due to the naive distribution of the protein sequences. SGEMM scalability is largely bound by interconnect performance; with the limited available memory, we cannot hide the transfer

	SMP	**Disk**	**Cluster Infiniband**	**Cluster GigE**	**Cluster PS3**	**Cell**
TaskCall	38	37	52	210	270	100
Alloc	13	3940	12	72.2	110	2.3
XferList	0.9	1.2	1.7	1.7	5.1	0.02
Xfer	0.7	31	13 (local) 30 (remote)	19 (local) 179 (remote)	35 (local) 382 (remote)	0.2
Barrier	88	N/A	221	315	610	0.3

Table 6.5: Overhead time in microseconds for the performance critical code paths. For the cluster runtimes, we include results for a transfer that involves data that is owned by the node (local) as well as data owned by a remote node (remote).

of the *B* matrix with computation. FFT3D is limited to network interconnect performance during the *Z* direction FFTs, similar to the other platforms. SAXPY and SGEMV are bound by M1-M0 DMA performance between the SPE's LS memory and node memory as well as runtime overheads. CONV2D are largely limited by the GigE interconnect when moving non-local portions of the image between nodes.

6.3 Runtime Overheads

In Table 6.5, we show the overhead of the critical parts of the runtime API for each runtime implementation measured in microseconds. **TaskCall** is the time for `CallChildTask` to execute an empty function on a single child and the immediate wait for the completion of the task (`WaitTask`). For the cluster runtime, this is the time to execute a task on a remote node. **Alloc** is the time it takes to allocate the smallest array possible for each runtime (`AllocArray`), a 16 byte array for the Cell runtime and a 1 byte array for all others, add the array to the system (`AddArray`), remove the array from the system (`RemoveArray`), and release any resources created (`FreeArray`). **XferList** is the time to create list via `CreateXferList` with a single entry to transfer the smallest array possible between the parent and child and as well as the time to release any resources created (`FreeXferList`). **Xfer** is the time to transfer the smallest array via a `Xfer` call and immediately wait for the transfer to complete (`WaitXfer`). For the cluster runtime which allocates a distributed array, we provide values for the transfer time if the results are local to the node as well as the cost of a remote access. For the disk runtime, all transfers occur from the file cache to differentiate the runtime overhead from the disk access latency. **Barrier** measures the time to complete a barrier across all children in the largest test configuration, i.e. 16 children for the cluster.

In general, our overheads are low, but non-trivial, for different API calls. For the SMP runtime, the largest overheads come from the cost of a task call and return, 38 microseconds, and the cost of a barrier, 88 microseconds. In the case of a task call and return, the overhead comes at the cost of creating a task execution request object and enqueuing it onto the task queue of the specified child. The parent has to lock the child queue, enqueue the object, and signal the waiting child thread. The child then has to wake up, recover the lock, run the function, and then signal the parent for completion. This cost is on the order of 100,000 cycles. A barrier requires similar locking and signaling, but there are eight processors vying for the mutex instead of just two. This increases the overhead to close to 250,000 cycles.

For the disk runtime, there is a disproportionate cost of an allocation, 3,940 microseconds – two orders of magnitude more expensive than other calls. This massive overhead, millions of cycles, is caused by the cost of creating a file on disk. However, allocations in the root of the hierarchy are rare for our applications and allocations are generally not in the critical execution path in optimized programs. `XferList` creation is slightly more expensive than on the SMP runtime because, instead of just storing pointers and offsets, we have to allocate and fill in structs for the Async I/O API. Since data transfers in the disk runtime rely on the Async I/O API in the Linux kernel, the cost of `Xfer` is much larger than most other runtimes. We have to post the asynchronous transfer request to the kernel and then wait for the transfer to complete and receive a signal from the kernel. It should be noted that we have constructed the test to read from the file cache and not access the disk, just traverse all of the runtime and system calls. Since the disk runtime has a single child, barriers are not applicable for this runtime.

For the cluster runtime, we present results for our implementation running on our different base configurations, the 16 node Infiniband cluster, the 4 node Gigabit Ethernet x86 based cluster, and a 2 node Sony Playstation 3 cluster. As expected, the overheads on the Infiniband cluster are much lower than on the other configurations using Gigabit Ethernet. In fact, our Infiniband implementation provides 5.8 microsecond latency for single byte messages and bandwidths of over 400 MB/s for large messages. The Gigabit Ethernet cluster has a latency of 126 microseconds and the PS3 cluster has a latency of 198 microseconds. The measured overhead cost of data transfers listed in Table 6.5 includes these network latencies. The differences in latencies are reflected in our overheads as we use small messages for communicating with nodes for task execution, barriers, and transfers. When requesting a data transfer, there are overheads for communicating between the communication and execution threads to initiate and wait for transfers as well as overheads for comparing the requested data against the interval tree to detect which node(s) we have to communicate with. When data transfers involve remote nodes, we incur all the overheads to setup and perform single sided communication (gets and puts) to other nodes. When data transfers are local to a node (e.g. the local node owns the portion of the distribution data transfers are being requested with) there are overheads in performing the pointer manipulation to prevent data copies. Barriers are much more costly on our cluster implementation than on others because our implementation relies on all nodes participating in the barrier sending messages to the other nodes in the barrier, as well as the communication thread and execution thread using mutexes to communicate. Barrier performance degrades as the underlying communication latency increases from the Infiniband cluster to the Playstation 3 cluster.

For the Cell implementation, calling a task on a child is orders of magnitude more costly than others. The largest overhead factor is the loading of the task overlay into the SPE. With SDK 2.1 and our overlay loading code, more than half of the overhead, 64 microseconds, is spent loading the code and starting execution of the task. The next largest overhead is comprised of the communication via mailboxing with the PPE for starting the task and notification of completion, 33 microseconds. The rest of the overhead, around 3 microseconds, is from the transfer of data from PPE to SPE required for execution such as pointers to arrays in the array table and LS addresses of the other SPEs for signaling. The runtime API calls involving transfer list creation and data transfers map very closely to calls provided in the Cell SDK minimizing overheads.

As runtimes are composed for more complex machines, the overheads of each runtime compose. For example, compare the overheads of the SMP runtime (Figure 6.1) and the cluster runtime (Figure 6.3) against the composed cluster of SMPs configuration (Figure 6.14) for different applications. The overheads in the composed machine are roughly the sum of the overheads of the individual runtimes. As seen in Figure 6.16, there is a notable exception to this behavior with SAXPY and SGEMV applications when running on the cluster of Sony Playstation 3's. The overheads of the Cell runtime are much larger when running with the cluster runtime as compared to running alone. This is caused by an interaction of cluster runtime with the memory system, causing the initiation of transfers between PPE and SPE, barriers, and overlay loading to take longer than expected. This is viewed as a bug in the implementation of the Cell SDK and MPICH-2 libraries we are using.

Chapter 7

Discussion

7.1 Machine Abstraction

We have shown that with our machine abstraction, we can efficiently map onto several common machine configurations, notably the Cell processor, a cluster, and a SMP. Furthermore, we can run on composite machines comprised of these architectures. One unique feature of this abstraction is that we can treat disk systems in the same manner as other parts of the memory hierarchy providing a uniform abstraction which includes I/O devices. Our abstraction allow us to capture the performance critical aspects of a machine: efficient use of the memory hierarchy and parallel execution.

One of the requirements of our abstraction is that a machine must be mapped into a tree hierarchy. While this matches some machines well, like those presented in this dissertation, other architectures may not have an efficient representation as a tree of memories. A common configuration that is hard to map into a tree abstraction is a cluster with a distributed disk system. If we treat the distributed disk system as a virtual level representing the aggregate disk system with a child for each disk, there is no way to treat the aggregate of the cluster memory as a single memory module as this is not a tree structure, e.g. the aggregate disk is a fan-out but the single aggregate cluster memory requires a fan-in. For such a machine, we would have to limit the representation to the aggregate of the disk memory and not model the aggregate of the cluster memory, potentially forgoing the ability to use the high speed network interconnect for node to node communication, and limit communication to be through the distributed disk interconnect. However, in many configurations like this, the disks are within each of the nodes or the distributed disk system uses the same interconnect as node to node communication, making the node interconnect and the disk interconnect are the same.

Complex interconnect topologies are also difficult to model using a tree based abstraction. Although we can naturally model tree based interconnection networks for mesh, ring, torus, hypercube and other topologies, we have to either model the processors as having equal communication distance, or only capture a portion of the interconnect capabilities. For example, if we attempt to model a 1-D mesh network as a tree, we would split the processors in left and right halves all the way down to pairs of processors. Although processor $N/2$ and processor $N/2+1$ have a physical connection, that connection cannot be realized in a tree abstraction.

7.2 Portable Runtime System

As we have shown in Chapter 6, our runtime system allows us to execute unmodified Sequoia applications on a variety of platforms achieving an unprecedented degree of machine portability. Our runtimes have low overheads, demonstrating that the machine abstraction and runtime interface allow for efficient implementations and allow applications to maximize bandwidth and computational resource utilization of the machine. The runtime interface is also simple with only 18 entry points, leading to rapid implementation. The cluster and Cell runtime implementation were the most complex, but unoptimized implementations were up and running within a few man weeks. The SMP and disk runtime implementations were only a few man days. Since the runtimes could be composed, there was no development time when moving to more complex machines.

Since the runtime system is designed to closely match the machine abstraction we have chosen, it does not export non-portable, but potentially useful, hardware features. An example of this is that the runtime on the Cell processor does not expose sibling communication even though the hardware is capable of supporting it and it can provide a much higher performance communication mechanism (204.8 GB/s SPE to SPE transfers vs. 25.6 GB/s for SPE to memory transfers). In the current system, to use this functionality, a runtime would need to be added to abstract sibling to sibling communication through a virtual level. The overhead for managing a virtual level can be costly, leading to efficiency problems for applications requiring SPE to SPE communication for performance.

The design of the presented runtime does not currently export portable high-level primitives, such as reduction or parallel prefix scan operations, that might be able to make use of specialized hardware features. For example, some systems like Blue Gene/L have custom reduction networks. If the runtime system presented some of these high-level operations as part of the interface, instead of relying on the compiler to generate code for these operations, it would be possible to take advantage of specialized hardware or optimized implementations available on each machine. However, adding higher level functionality is a slippery slope and makes the interface more complex and more difficult to implement.

The current runtime system cannot handle variable computational load dynamically. Currently, the runtime system schedules tasks as instructed by the caller. However, all the runtimes are currently implemented using task queue structures. Instead of assigning tasks explicitly on the children, the parent could instead enqueue tasks on a work queue and allow idle processors to schedule work. Another option would be to use task queues per child and allow children to steal work from other children.

7.3 Sequoia

Sequoia introduces the notion of a hierarchical memory directly into the programming model to increase both the performance and portability of applications. Sequoia programs describe how data is moved and where it resides in a machine's memory hierarchy along with how computation is performed via tasks. Tasks are an abstraction for self-contained units of communication, working set, and computation. Tasks isolate each computation in its own local address space and express parallelism. To enable portability, Sequoia maintains a strict separation between algorithm description and machine specific optimization.

While Sequoia excels on regular applications or applications that can be easily regularized, the Sequoia programming model can make it difficult to express several types of applications. Applications with irregular data access patterns can be difficult to express in Sequoia. In a two-level machine, a child is capable of issuing read requests from the highest memory level array, thus being able to read from any memory location. However, in a multi-level machine, the leaf can only read from its parent, but its parent may only have a portion of the entire data. Thus, the only practical way to run irregular computations that span multiple physical address spaces is to regularize the application.

Some applications are very difficult to regularize and achieve efficient execution. For example, ray-tracing relies on the traversal of a spacial subdivision structure, often a kD-tree, for efficiency. For primary rays, it is possible to perform a screen based subdivision and calculate the required sub-tree for each ray in the scene, but these sub-trees may be of variable size. Where things become difficult is in the behavior of secondary rays. A ray that reflects of a surface may require very different parts of the acceleration structure that the parent in a multi-level system may not have. In this case, the Sequoia model forces the user to return to the parent, saving off required state, and performing computation on the parent to figure out what data is required by its children, attempt to fetch the needed data and then return execution to the child. If control must be returned all the way up to the root of the hierarchy, the saved state required to restart all of the children may be quite large. Applications like raytracing which assume global memory access would have to be reformulated to execute efficiently in Sequoia. However, applications like this cannot run on any architecture without shared memory support, and as shared memory machines provide progressively less uniform memory access, these algorithms will need to be reformulated to achieve efficient performance.

Applications requiring variable output are also difficult to express in Sequoia. Just like applications which require irregular input must provide a maximum size and the runtime will allocate for this worst case, variable output must also allocate for the largest possible size. However, since writes to this array occur in parallel, without knowing ahead of time how much data will be written, in the general case the user will end up with gaps in the final output. As such, either the user must manage this behavior by tracking how much data was written to each block in order to allow them to generate indexes with which to read the data as input, or the user/runtime has to compact the data on the return of control to the parent. Sequoia does not currently define the semantics of this behavior and currently performs no compaction on the data.

Mutable data structures are also difficult to express in Sequoia. The hierarchical nature of Sequoia makes global data structure manipulation difficult. For example, consider insertion into a balanced tree structure. A leaf task may insert an element into the tree that causes a rotation about the root of the tree. This rotation will affect the data structure as viewed by all hierarchy nodes. The only way to perform this update would be to return control all the way up to the parent, but Sequoia provides no mechanism to inform all tasks complete and return control to the parent. Furthermore, on some machines which do not have a control processor at the root that can access the entire global memory, a disk system for example, the user *must* provide a way to decompose their data access into arrays that can fit in the child's memory for execution.

Large applications can be difficult to manage in Sequoia because of the mapping and tuning requirements. All the applications in this dissertation were mapped to and tuned for each machine by hand. For large applications, this can be daunting, especially when it comes to the interaction between producer/consumer pairs or space restrictions in the memory levels. We have begun recent work into automatically mapping applications to a machine and performing a search over tunables to improve performance and handle large applications [Ren et al., 2008]. This auto-tuning framework fits nicely into the Sequoia model and makes the mapping and tuning of large applications more approachable.

7.4 Future Work

There are also other systems for which it would be useful to develop an implementation of our API. For example, GPUs use an explicit memory management system to move data and computational kernels on and off the card. The BrookGPU system [Buck et al., 2004] has a simple runtime interface which can be adapted to our interface. Having an implementation of our runtime for GPUs would, in combination with our existing runtimes, immediately enable running on multiple GPUs in a machine, a cluster of nodes with GPUs, and other, more complex compositions of GPU systems. However, it should be mentioned that generating efficient leaf tasks for GPUs is non-trivial; our runtime and system would aim to solve data movement and execution of kernels on the GPUs, not the development of the kernels themselves.

Scalability to very large machines, which we have not yet demonstrated, is future work. Previous successful work on distributed shared memory implementations for large clusters can be adapted to our runtime system. Dealing with load imbalance is also a problem for the current implementation. However, since our runtimes use queues to control task execution, adapting previous work on work-stealing techniques appears to be a promising solution, but will require support from the compiler for dynamic scheduling of tasks by the runtime and consideration of the impact of rescheduling tasks on locality as discussed in Blumofe et al. [Blumofe et al., 1995] and explored further in Acar et al. [Acar et al., 2000]. Scaling to machines with many more processors as well as even deeper memory hierarchies is the next goal of this work.

Research into transactional memory has great promise to improve the behavior and cost of synchronization and allowing for efficient manipulation of mutable data structures in parallel [Herlihy and Moss, 1993; Shavit and Touitou, 1995; Harris et al., 2005; Carlstrom et al., 2006]. It would be interesting to explore combining the efficient memory hierarchy behavior of Sequoia with transactional memory. Since a task encompasses the data and execution environment of a task, as simple way to combine the techniques is to wrap a task in a transaction and re-execute the task if rollback is required. This would remove the need for Sequoia to forbid write aliasing, allowing for more flexible application design, as well as providing for hierarchical transactions. An unexplored but potentially interesting avenue of research is the effect of adding support for transactions directly into the Sequoia model.

This would potentially give programmers the full flexibility of transactions in a general way that may be memory hierarchy aware but still allow for the efficient expression of parallelism and communication through the memory hierarchy.

Chapter 8

Conclusion

This dissertation has presented a runtime system that allows programs written in Sequoia, and more generally in the parallel memory hierarchy model, to be portable across a wide variety of machines, including those with more than two levels of memory and with varying data transfer and execution mechanisms. Utilizing our runtime abstraction, our applications run on multiple platforms without source level modifications and maximize the utilization of available bandwidth and computational resources on those platforms.

One of the most interesting features of our design is that virtualization of all memory levels allows the user to use disk and distributed memory resources in the same way that they use other memory systems. Out-of-core algorithms using disk fit naturally into our model, allowing applications on memory constrained systems like the Sony Playstation 3 to run as efficiently as possible. Programs can make use of the entire aggregate memory and compute power of a distributed memory machine using the same mechanisms. And, despite the explicit data transfers in the programming model, through a contract between the runtime and compiler we also run efficiently on shared memory machines without any extra data movement.

All of our runtimes are implemented using widely used APIs and systems. Many systems, like those underpinning the languages and runtime systems from Section 2.2, could be adapted relatively easily to support our interface. Conversely, our interface and implementations are also easily adaptable for systems that use explicit memory transfers and control of parallel resources. And, although we have presented the runtime as a compiler target, it can also be used directly as a simple programming API.

8.1 Thesis Summary

First uniform interface for multi-level memory hierarchies We have described a uniform runtime interface that provides mechanism independence for communication and thread management. This allows us to abstract SMPs, clusters, disk systems, Cell blades, and a Sony Playstation 3 with the same interface. Furthermore, we can abstract complex machines through composition of runtimes allowing for program execution on a cluster of SMPs, a Sony Playstation 3 pulling data from disk, and a cluster of Sony Playstation 3's.

Simple runtime interface The runtime interface has only a few entry points that require implementation for supporting the abstraction. Runtimes can choose to extend and add features for data layout/distributions or more complex execution, but the base interface is simple and much of the interface is easy to implement on many machines using standard APIs.

Code portability By using the proposed runtime system, we have shown unmodified Sequoia programs running on multiple machines by just changing the runtime being called. We have also shown that along with a compiler, we can compose runtimes and run the same Sequoia programs on machines with more complex memory hierarchies.

8.2 Observations

There have been many lessons learned throughout this research project. Exploring algorithm implementations across different architectures show common performance traits. Performance oriented programming on all architectures primarily entails careful management of communication through the machine and the exploitation of parallel computational resources. Memory performance is becoming such a large performance bottleneck on large machines that in practice, even if shared memory semantics are available on a machine, performance oriented programmers abandon shared memory programming on large shared memory machines because of the non-uniform memory access performance. Instead the programmers are switching to distributed memory techniques and explicitly controlling data layout and access. On clusters, programmers spend a large amount of time refactoring their code and algorithms to minimize communication. The issue is that programmers generally do not start with the idea of minimizing communication cost, and sometimes doing so requires a different algorithm than originally chosen. Often back-porting code from the Cell or cluster implementation will outperform the original code on a shared memory machine because the programmer has been forced to focus on limiting communication.

The cost of synchronization and data transfers has become a dominant factor in the performance of many algorithms. Many programming languages and models have focused on expressing parallelism but have not concerned themselves with helping the programmer to describe the transfer of data through the memory hierarchy. As the difference in bandwidth and latency between hierarchy levels continues to grow, the problem of efficiently managing data movement through the machine becomes paramount. On parallel systems, many applications are limited not by computation but instead memory bandwidth. Since efficient use of the memory system is of such great importance to performance, Sequoia was designed to make the best use of the memory system as possible by forcing the programmer to express how they are using data so that communication can be structured as efficiently as possible. Moving forward, we hope that future languages continue to emphasize structured communication as well as computation and explore ways to handle synchronization and irregular applications.

It is important to note that we are beginning to shift towards consumer parallel computing on a large scale. Both AMD and Intel are shipping quad core processors today and have roadmaps showing processors with greater than 64 cores by 2015. GPUs are already starting to be used for tasks other than graphics because of their processing capabilities, 0.5 TFLOPS in the current generation. Efficient GPGPU programs are massively data-parallel, but newer architectures are gaining synchronization and communication capabilities. Game consoles developers are already contending with parallel programming environments with heterogeneous cores. It is unclear whether the future of parallel processing will be in homogeneous designs like Intel's Larabee or in heterogeneous designs like AMD's Fusion, but the fundamental issue for programmers will be how to efficiently manage the memory hierarchy and parallel processing resources. However, each new parallel architecture has come with its own programming environment. The overwhelming mix of architectures and programming models make it difficult for a developer to produce and ship products across a wide variety of platforms. Better programming models are needed that provide abstractions that work across as many architectures as possible, reducing or eliminating the porting effort that can swamp developers and limit application support to a subset of consumer systems.

It is my belief the largest fundamental problem in computer science is that the use common data structures and the data access behavior of common algorithms are not well understood in the face of parallelism. Many programmers start with heavily optimized sequential algorithms and attempt to port them to parallel machines and are dismayed by the scaling performance. Moreover, most programmers and computer scientists focus heavily on using data structures which were designed without parallelism in mind. Most parallel data structures are built from sequential data structures and rely on fine-grain locking semantics to provide correctness that do not scale well to large machines. Transactional memory looks promising in this area, but this research area is still young and much of the effort as focused on commonly used data structures and algorithms and there has not been much exploration into new data structures and algorithms that maximize the benefits of transactional memory. We hope that more research begins to go into new parallel algorithms and data structures and that the computer science curriculum begins to require undergraduates take courses in parallel programming (in *any* well established language) and parallel data structures.

8.3 Last Words

In summary, the Sequoia environment including the language, compiler, and runtime system provides a way to design and run applications that maximize the utilization of machine resources while allowing for portability across many machine configurations. We have shown that an interface can be designed to capture performance critical aspects of many common architectures while abstracting the different architecture mechanisms with low overhead. Our hope is that many of the ideas behind the system will influence other parallel programming languages and runtimes in the way they handle expressing the memory hierarchy. Sequoia makes the programmer carefully think about how to decompose their computation and data into tasks and think about how data is being used and manipulated in their algorithms. If we continue on the current trend, architectures will gain ever deeper and more complex memory hierarchies in order to bridge the gap between computational performance and memory bandwidth, and we will gain ever increasing numbers of parallel processing elements. For applications to increase performance, programmers will have to (re)design algorithms that can make the best use of the memory hierarchy and scale with increasing processor counts, and we need programming models and abstractions that assist the programmer in this task.

Bibliography

[Acar et al., 2000] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[Aho et al., 1974] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Aho et al., 1986] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Alpern et al., 1993] B. Alpern, L. Carter, and J. Ferrante. Modeling Parallel Computers as Memory Hierarchies. In *Proc. Programming Models for Massively Parallel Computers*, 1993.

[Alpern et al., 1994] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 12(2/3):72–109, 1994.

[Alpern et al., 1995] B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: A methodology for portable high performance. In *International Working Conference on Massively Parallel Programming Models*, 1995.

[AMD, 2007] AMD. HD2900XT (R600). http://ati.amd.com/products/radeonhd2900, 2007.

[ANL, 2007] ANL. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2, 2007.

[Bilardi et al., 1996] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul G. Spirakis. BSP vs LogP. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, 1996.

[Blumofe et al., 1995] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[Buck et al., 2004] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[Callahan et al., 2004] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60. IEEE Computer Society, 2004.

[Carlson et al., 1999] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. University of California-Berkeley Technical Report: CCS-TR-99-157, 1999.

[Carlstrom et al., 2006] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos Transactional Programming Language. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. Jun 2006.

[Carmean, 2007] Doug Carmean. Intel Larrabee - Stanford CS448 presentation, May 2007.

[Chow et al., 2005] A. Chow, G. Fossum, and D Brokenshire. A Programming Example: Large FFT on the Cell Broadband Engine, 2005.

[Cormen et al., 2001] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 311–317. McGraw-Hill, 2001.

[Culler et al., 1993] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[Dally et al., 2003] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 35, Phoenix, Arizona, 2003.

[DARPA, 2007] DARPA. HPCS High Productivity Computer Systems. http://www.highproductivity.org/, 2007.

[Deitz et al., 2004] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Abstractions for dynamic data distribution. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 42–51. IEEE Computer Society, 2004.

[Eager and Jahorjan, 1993] Derek L. Eager and John Jahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Trans. Comput. Syst.*, 11(1):1–32, 1993.

[Fatahalian et al., 2006] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[Fortune and Wyllie, 1978] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. ACM STOC*, pages 114–118, 1978.

[Forum, 1993] High Performance Fortran Forum. High performance

Fortran language specification. *SIGPLAN Fortran Forum*, 12(4):1–86, 1993.

[Frigo et al., 1999] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[Frigo, 1999] Matteo Frigo. A fast Fourier transform compiler. In *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pages 169–180, May 1999.

[Fukushige et al., 2005] T. Fukushige, J. Makino, and A. Kawai. GRAPE-6A: A Single-Card GRAPE-6 for Parallel PC-GRAPE Cluster Systems. *Publications of the Astronomical Society of Japan*, 57:1009–1021, dec 2005.

[Geist et al., 1994] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing. Cambridge, MA, USA, 1994. MIT Press.

[Harris et al., 2005] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel* *programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[Herlihy and Moss, 1993] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support For Lock-Free Data Structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[Horn et al., 2005] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[Houston et al., 2008] Mike Houston, Ji Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. A Portable Runtime Interface For Multi-Level Memory Hierarchies. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2008.

[Huang et al., 2006] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand. In *International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2006.

[Ian Buck, 2003] Ian Buck. Brook Specification v0.2. http://merrimac.stanford.edu/brook/, Oct. 2003.

[IBM, 2007a] IBM. IBM BladeCenter QS20. http://www-03.ibm.com/technology/splash/qs20, 2007.

[IBM, 2007b] IBM. IBM Cell Broadband Engine Software Development Kit. http://www.alphaworks.ibm.com/tech/cellsw, 2007.

[Intel, 2005] Intel. Math Kernel Library. http://www.intel.com/software/products/mkl, 2005.

[Kalé and Krishnan, 1993] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[Knight et al., 2007] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for Explicitly Managed Memory Hierarchies. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 226–236, March 2007.

[Labonte et al., 2004] Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The Stream Virtual Machine. In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*, Antibes Juan-les-pins, France, September 2004.

[LANL, 2008] LANL. Roadrunner. http://www.lanl.gov/orgs/hpc/roadrunner/, 2008.

[MPIF, 1994] MPIF. MPI: A message passing interface standard. In *International Journal of Supercomputer Applications*, pages 165–416, 1994.

[MPIF, 1996] MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[Numrich and Reid, 1998] Robert W. Numrich and John Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[NVIDIA, 2007] NVIDIA. CUDA. http://www.nvidia.com/CUDA, 2007.

[Owens et al., 2008] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5), May 2008.

[Ren et al., 2008] Manman Ren, Ji Young Park, Timothy Knight, Mike Houston, Alex Aiken, William J. Dally, and Pat Hanrahan. A Tuning Framework For Sofware-Managed Memory Hierarchies. Feb. 2008.

[Shavit and Touitou, 1995] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[Sony, 2007] Sony. Sony Playstation 3. http://www.us.playstation.com/PS3, 2007.

[Su et al., 1985] Weng-King Su, Reese Faucette, and Charles L. Seitz. The C Programmer's Guide to the COSMIC CUBE. CaltechCSTR:1985.5203-tr-85, 1985.

[Valiant, 1990] L. G. Valiant. A Bridging Model for Parallel Computation. *CACM*, pages 103–111, August 1990.

[Yelick et al., 1998] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, 1998.