This Dissertation

entitled

EXPLOITING LOCALITY WITH QTHREADS FOR PORTABLE

PARALLEL PERFORMANCE

typeset with NDdiss2$_\varepsilon$ v3.0 (2005/07/27) on October 21, 2009 for

Kyle Bruce Wheeler

This LaTeX 2$_\varepsilon$ classfile conforms to the University of Notre Dame style guidelines established in Spring 2004. However it is still possible to generate a non-conformant document if the instructions in the class file documentation are not followed!

> Be sure to refer to the published Graduate School guidelines at `http://graduateschool.nd.edu` as well. Those guidelines override everything mentioned about formatting in the documentation for this NDdiss2$_\varepsilon$ class file.

It is YOUR responsibility to ensure that the Chapter titles and Table caption titles are put in CAPS LETTERS. This classfile does *NOT* do that!

This page can be disabled by specifying the "`noinfo`*" option to the class invocation.* (i.e.,`\documentclass[…,noinfo]{nddiss2e}`)

This page is *NOT* part of the dissertation/thesis, but MUST be turned in to the proofreader(s) or the reviwer(s)!

NDdiss2$_\varepsilon$ documentation can be found at these locations:

`http://www.gsu.nd.edu`
`http://graduateschool.nd.edu`

EXPLOITING LOCALITY WITH QTHREADS FOR PORTABLE PARALLEL PERFORMANCE

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Kyle Bruce Wheeler

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
October 2009

EXPLOITING LOCALITY WITH QTHREADS FOR PORTABLE PARALLEL PERFORMANCE

Abstract

by

Kyle Bruce Wheeler

Large scale hardware-supported multithreading, an attractive means of increasing computational power, benefits significantly from low per-thread costs. Hardware support for lightweight threads and synchronization is a developing area of research. Shared memory parallel systems are flourishing, but with a wide variety of architectures, synchronization mechanisms, and topologies. Portable abstractions are needed that provide basic lightweight thread control, synchronization primitives, and topology information to enable scalable application development on the full range of shared memory parallel system designs. Additionally, programmers need to be able to understand, analyze, tune, and troubleshoot the resulting large scale multithreaded programs.

This thesis discusses the implementation of scalable software for massively parallel computers based on locality-aware lightweight threads and lightweight synchronization. First, this thesis presents an example lightweight threading API, the qthread library, that supports the necessary features in a portable manner. This exposes the need for a structural understanding of parallel applications. ThreadScope, a tool and visual language for structural analysis of multithreaded parallel programs is presented to address this need. A strong

understanding of algorithm structure combined with a locality-aware portable threading library leads to the development of three distributed data structures—a memory pool, an array, and a queue—that adapt to system topology at runtime. Such adaptive data structures enable the development of three example adaptive computational templates—sorting, all-pairs, and wavefront—that hide the parallelism details without sacrificing scalable performance.

To my ever-loving wife, Emily, and everyone who has had patience with me and confidence in me. And to our daughter, Julia, for reminding me why I needed to graduate in the sweetest way possible.

CONTENTS

FIGURES

ACKNOWLEDGMENTS

This work was funded in part by Sandia National laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Modern supercomputers have largely taken the route of parallel computation to achieve increased computational power [3, 7, 18, 51, 78, 101, 176]. Increased parallelism leads to increased complexity, and is more difficult to program. In many modern supercomputers, shared memory has been traded for distributed memory because of the cost and complexity involved in scaling shared memory to the size systems needed for supercomputing applications. This design is vulnerable to an obvious bottleneck: the communication between parallel nodes.

The most persistent and troublesome problem with communication in shared memory systems has been latency: the processor is faster than the storage mechanism and must wait for it to return data for use in computation. Whether the disparity is high-speed vacuum tubes waiting for high-density (low speed) drum memory, or high-speed CPUs waiting for high-density (low speed) DRAM, the problem is the same. This is a fundamental vulnerability in the von Neumann computational model, because of the separation of logic from storage and their dependence on one another. Computation can only proceed at the pace of the slower of the two halves of the computational

model. The slower has historically been the memory, so the problem is generally termed "memory latency" or the "memory wall". Memory latency is particularly problematic with large data-dependent computation [115, 152, 183].

There are only two fundamental approaches to addressing the problem of memory latency: either tolerate the problem or avoid the problem. The "toleration" approach focuses on the overall throughput of the system, and masks the latency of requesting things from memory by performing unrelated useful work while waiting for each memory operation to complete. Examples of this idea include the work in Simultaneous Multithreading (SMT) [82, 118] and dynamic scheduling or out-of-order execution [75], both of which identify work that is sufficiently unrelated that it can be performed at the same time. The more unrelated or independent work that needs to be accomplished, the greater the ability to hide the latency involved in memory operations.

The other approach, "avoidance", generally uses specialized hardware to prevent latency effects. A common example of such specialized hardware is a memory cache, which provides a small and very fast copy of the slower high-density memory. The fundamental characteristic of computation that allows caches to work well is temporal locality: the smaller the set of data in use at a time, the more easily it can be stored in a small, fast cache. As caches are smaller than main memory, and generally smaller than most data sets used in computation, they do not solve the general problem, but do avoid it to the extent that the working set of information for any contiguous subset of the program's instructions can fit within the cache.

1.2 Problem

In 1946 Von Neumann himself, with Burks and Goldstine, recognized the problem of latency and proposed the cache-based avoidance technique [21] in a paper that is commonly cited in computer architecture texts [75, 140]:

> Ideally one would desire an indefinitely large memory capacity such that any particular ... [memory] word ...would be immediately available. ...It does not seem possible physically to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

As predicted, this bottleneck has been a powerful consideration in computer design and architectural research for the past six decades, with each new computer architecture seeking to address the problem in one way or another. Despite this effort, the problem has only gotten worse with time. Figure 1.1 depicts the progression of the problem in simple terms, based on information from Intel's processor line, starting in 1971 with the 4004 and continuing through the Pentium IV in 2006.

Caching is one of the most well-known and well-studied ways of addressing the von Neumann bottleneck. As such, it has drastically affected the design of compilers and software by providing a simple metric for optimizing performance: temporal locality. To increase temporal locality, memory "hot spots" are created intentionally by compilers and software designers so that frequently used data will fit within the small amount of fast memory available. While hot spots have proved extremely successful in simple von Neumann single-processor designs, supporting coherent caches in parallel systems places a significant burden on memory bandwidth [54]. Worse, the more nodes in the parallel system, the more expensive this cache coherency overhead be-

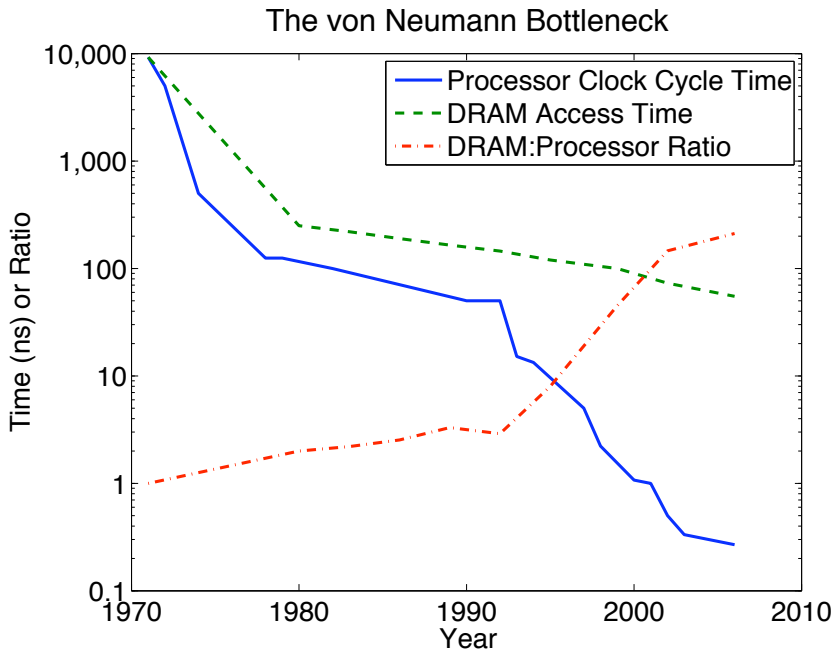


Figure 1.1. The von Neumann Bottleneck. [122]

comes. Because cache coherency has such high overhead in parallel systems, typical parallel software avoids sharing data implicitly and at a fine-grained level. Instead, data is commonly shared explicitly and large chunks of it at a time [61]. This leads some supercomputer designers to do away with processor cache entirely in large parallel shared memory computers [38], a design decision which invalidates the wisdom of creating memory hot-spots. However, because of the physical reality that in a large enough system not all memory addresses have the same latency, randomly distributing data ranges across all memory banks [38] does not take advantage of what ranks of locality naturally exist in hardware design. Additionally, this decision removes a major tool for addressing the problem of the memory wall.

Large-scale shared-memory computers that expose locality are typically referred to as cache-coherent non-uniform memory access (ccNUMA) systems. This is a reference to two fundamental aspects of the design of such systems. First, while they have multiple processors, each processor has its own "local" block of memory that is accessible from all processors and whose effective access latency depends on the relative location of the accessor. The latency is lowest when accessing that memory from the nearest processor, higher when accessing from nearby processors, and higher still from processors further away. Thus, the ratio of processor speed to memory latency is significantly higher for more distant blocks of memory than for more local blocks. Second, each processor has a cache hierarchy, and these caches work together to provide a coherent image of the state of memory. Maintaining this coherence has a significant overhead, and serves generally to increase the latency of memory operations. Due to the prohibitive complexity of programming shared-memory systems without a coherent view of memory, coherency is generally regarded as necessary. Historically, memory latency in ccNUMA machines has been a low-priority issue [132] because the range of latencies has been relatively low. However, as ccNUMA machines have gotten larger the range of memory latencies has increased. Additionally, these variations have become more important as the processor-memory speed ratio has increased, allowing more and more processor cycles fit into a given time slice and thereby significantly increasing the relative penalty imposed by non-local memory accesses.

Some of the problems inherent in current ccNUMA systems may be avoided with a new architecture. One way of reducing memory latency is to integrate

the processor with its memory. If local memory access is sufficiently fast, the system does not require a cache, and thus can avoid the overhead associated with maintaining cache coherency in a parallel system. This idea has been examined by projects such as the EXECUBE [98] and its processor-in-memory (PIM) successors [18, 19, 99, 123], IRAM [141], Raw [179], Smart Memories [111], Imagine [147], FlexRAM [93], Active Pages [138], DIVA [69], Mitsubishi's M32R/D [135], and NeoMagic's MagicGraph [126] chips, to name a few.

The major attraction of integrating memory and processor is that it brings the data much closer to where it will be used, and thus provides the opportunity for both increasing bandwidth and decreasing latency. The PIM design concept attacks the von Neumann bottleneck directly; by moving the processor closer to the memory—avoiding cache, bus, and memory-controller overhead—the latency of accessing local memory is decreased. The use of many small processors rather than a single central processor also increases the potential for parallel execution, which improves tolerance of memory latency as well.

It is conceivable to build a shared memory parallel computer almost entirely out of PIM units. Such a computer would have an extremely large number of parallel computational nodes, however each node would likely not be as fast or as complex as a modern central processor. This unusual architecture would place unusual demands on its software and on the developers who write that software, in large part because the standard issues of scheduling and data layout are combined.

In a NUMA system as potentially dynamic as a PIM-based shared memory system, placement of data becomes not merely a crucial problem, but a problem for which the optimal answer may change over the course of a given set of computations. Of course, commodity systems are not generally non-uniform, and do not provide all of the same hardware-based parallelism features that larger systems do. Thus, developing software for large shared-memory systems frequently requires access to those systems. This aspect of the general programming problem is the target of this thesis:

> *How can programmers create applications that take full advantage of new hardware-accelerated parallelism features in large-scale shared memory environments and topologies without sacrificing portability or performance on commodity systems?*

1.3 Contribution

The area of cache-aware data-layout is a well-researched area [12], beginning even before memory hierarchies were first introduced in the early 1950's [149]. At the time, programmers needed to manage the memory hierarchies themselves, explicitly copying data from drum to core memory and back. In 1962, Kilburn et al. [95] proposed automatic management of core memory, which was first implemented in the Atlas computer. Similarly, as a skilled parallel programmer may well be able to plan data layout for large programs and manage it to provide optimal performance on any size system, it is both more convenient and more portable to do this automatically, both as is currently done at compile time and as this thesis demonstrates, at runtime. While runtime reorganization was proposed and demonstrated on serial computation by Ding et al. [48], and data-agnostic page-based memory mi-

gration has been thoroughly studied [133], coordination between scheduling decisions and data layout based on computational patterns has not been investigated. In addition to convenience and portability, runtime adaptation to system topology enables the exploitation of domain-specific knowledge while reducing the amount of detailed knowledge of the system required to leverage the hardware layout of any particular large NUMA system.

The primary contribution of this work is a scalable, portable, and flexible approach to fine-grained large-scale data and computational locality that adapts to provide performance on a wide range of NUMA topologies. This approach has four basic components: the qthreads lightweight threading interface, an examination of algorithm structure via ThreadScope, three basic data-parallel distributed data structures, and three example parallel computation templates.

1.4 Approach and Outline

In order to adapt to multiple topologies and fully exploit available locality, software must be able to locate both data and computational threads in specific locations, and must be able to determine the latency penalties of communications between locations. The goal of careful data and thread placement is to minimize the number of remote memory operations. In order to provide the greatest flexibility to fully exploit runtime adaptation of computation and data, it is necessary to expose as many parallel operations as possible in a given set of code so that each independent operation can potentially be executed in the optimal location. Specifically, a programming model that supports lightweight threading is necessary to allow programmers to express

the full measure of parallelism in their algorithms. This threading model must make locality a first-class aspect of each thread, so that the location of a thread can be used to make decisions about the work that it will do. Existing lightweight threading models explicitly avoid making locality guarantees and so a new lightweight threading interface that meets this requirement, the qthread interface [182], is presented in Chapter 3. This chapter also presents a library-based UNIX implementation of the interface.

The next step in leveraging machine layout is understanding the structure of the algorithms being used. Determining algorithm structure, independent of hardware, is a daunting task. ThreadScope [181], a visualization technique tailored for a lightweight threading environment, is presented in Chapter 4. It uses existing tracing tools — including Dtrace [24], Apple's libamber [6], the SST simulator [148], or an instrumented version of the qthread library — to instrument multithreaded applications and uses those traces to visualize the logical structure. The logical structure of multithreaded programs does not rely on a specific order of execution other than that specified and enforced by synchronization methods.

Building parallel applications that exploit the design of a given parallel system is a complex task, parallel applications can be built upon simple parallel data structures. Basic parallel data structures, such as distributed arrays, are key tools in parallel software design. Several data structure designs, implemented in the new lightweight threading environment, are presented in Chapter 5. These data structures hide the complexity of data locality and enable the programmer to easily take advantage of a wide range of hardware topologies. This chapter presents the design of the "qpool", a distributed memory pool,

the “qarray”, a distributed array, and the “qdqueue”, a distributed end-to-end ordered queue. All three designs are demonstrated on three dramatically different shared memory architectures: a small four-core developer workstation, a high throughput 128-core server, and a 48-node ccNUMA machine. The qpool can be up to 155 times faster than the standard `malloc()` implementation and provides location-specific memory. The qarray supports strong scaling, providing a 31.2 times performance improvement with 32 ccNUMA nodes while iterating over its elements. At scale, the qdqueue demonstrates up to a 47 times improvement over a strictly-ordered lock-free queue, and an 8.3 times improvement over the performance of state-of-the-art concurrent queues on a large ccNUMA system.

The data structures presented in Chapter 5 are used to build larger application templates in Chapter 6. Three templates are presented: sorting, all-pairs, and wavefront. The sorting template demonstrates portable design choices in a parallel quicksort implementation, particularly for avoiding false sharing [53]. The all-pairs template demonstrates an adaptive technique for efficient distribution of localized input data among parallel tasks. The wavefront template demonstrates the use of such an adaptive distribution technique for chained computations.

CHAPTER 2

RELATED WORK

2.1 Threading

This research draws from several categories of prior work, including threading interfaces, data structures, methods of discovering locality, and others. Standard threading interfaces — such as POSIX threads (pthreads) [136] and Windows Threads [86, 178] — are heavyweight threads. They provide several features and guarantees that impose significant overhead. For example, a standard POSIX thread is normally able to receive asynchronous interrupt-based signals. Because multiple signals can arrive at the same time, there is an inherent vulnerability to race conditions which must be dealt with, generally through the use of a per-thread signal stack. In order to receive such signals, the thread must be interruptible, which requires the ability to save all current thread state at any time so that it can be restarted. To then handle signals, each thread must have an interrupt vector that can be triggered by the signal. All of this either requires an OS representation of every thread or requires user-level signal multiplexing [56]. The drawbacks of heavyweight, kernel-supported threads — such as POSIX and Windows threads — are well-known [9], leading to the development of a plethora of user-level threading designs. Threads in a large-scale multithreading context often do not require any of the expensive

guarantees and features that such heavyweight threads provide, such as per-thread process identifiers (PIDs), signal vectors, preemptive multithreading, priority-based scheduling, and the ability to remotely cancel threads, among others. Threads without these features and guarantees are, for the purposes of this thesis, dubbed "lightweight" threads.

2.1.1 Threading Concepts

The coroutine [33, 153] is a generalization of subroutines that is often considered to be a threading concept, though it is not truly parallel. As a generalization of subroutines, coroutines establish a type of virtual threading even in a serial-execution-only environment, by specifying alternative contexts that get used at specific times. Coroutines can be viewed as the most basic form of cooperative multitasking, though they can use more synchronization points than just context-switch barriers when run in an actual parallel context. Because they require few guarantees, coroutines can be quite lightweight. One of the more powerful details of coroutines is that one routine specifies which routine gets processing time next. The generalized form of this behavior is known as "continuations" [72, 116]. Continuations, in the most broad sense, are primarily a way of minimizing state during blocking operations. When using heavyweight threads, whenever a thread does something that causes it to stop executing, its full context—complete with its stack, signal stack, a full set of processor registers, and any other thread-specific data—are saved so that when the thread becomes unblocked it may continue as if it had not blocked. This is sometimes described as a "first-class continuation." However, the continuation concept is more general. In the most general form, the programmer

may specify that when a thread blocks, it actually exits. When the thread would have unblocked, a new thread is created with programmer-specified inputs. This requires the programmer to explicitly save any necessary state while disposing of any unnecessary state.

There are threading concepts that make even fewer guarantees. Protothreads [52, 68] assert that outside of the active set of CPU registers there is no thread-specific state at all. This makes them extremely lightweight but limits their utility and flexibility. For example, at most, only one of the active stackless threads can call a function. These limitations have repercussions for ease-of-use and have limited their adoption.

2.1.2 Lightweight Threads

Several user-level threading designs qualify as lightweight, but have one of two major drawbacks: they either are not designed for large-scale locality-aware threading, or they require a special compiler (or both). Python stackless threads [145], a protothreads design, are an example threading design with the first drawback. Putting aside issues of usability with their inflexible minimal state, which are significant, the interface allows for no method of applying data parallelism to the stackless threads: a thread may be scheduled on any processor. Many other threading designs, from nano-threads [37, 114] to OpenMP [137], similarly lack a sufficient means of allowing the programmer to specify locality.

Programming languages such as Chapel [23], Fortress [5], and X10 [29], or modifications to existing languages such as UPC [55], require special compilers. As such, they have capabilities that library-based approaches do not, such

as the ability to enforce implicit locality as a function of the programming environment and to automatically detect independent loop iterations. These languages can provide more convenient parallel semantics than approaches based on library calls, but break compatibility with existing large codes.

Cilk [16] and OpenMP [137] are examples of interfaces that provide relatively convenient lightweight threads. Cilk in particular uses a continuation-style approach that helps minimize state-based overhead. They are both modifications of C designed to make threading a basic feature of the language. However, they ignore data locality, forcing the programmer to rely on the operating system's scheduler to keep each thread near the data it is manipulating and on the operating system's page migration policies for keeping data close to the threads operating upon it. At the same time, however, they both use work-stealing scheduling algorithms that assume that all tasks can be performed equally well on any processor. This assumption, which is an invalid assumption on NUMA machines with a high variability in latency, undermines the efforts of the operating system to keep threads and their data close together and leads to poor scheduling decisions.

2.2 Application Structure and Visualization

In order to fully leverage locality and the available hardware, it is necessary to understand the structure of the algorithms being used in a way that is independent of any specific hardware. As Goldstein et al. [66] discussed, most threading models conflate logical parallelism and actual parallelism. This semantics problem typically requires that programmers tailor the expression of algorithmic parallelism to the available parallelism, thereby forcing pro-

grammers to either submit to unnecessary overhead when there is less parallel hardware than the software was designed for or forgo the full use of the available hardware when there is more available parallelism than was planned for. The essence of adapting complex software to equally complex hardware is bridging this divide efficiently. This bridge can only be made through understanding the structure of both hardware and software. While hardware can be complex, software algorithms are not bound by spacial dimensions and can be even more complex. Visualization is a common means of presenting complex information so that it can be understood.

Multithreaded applications have a history of being difficult to visualize, because there are few strict rules about their behavior. Some of the oldest parallel visualizers, such as Pablo [62] and Tapestry [112], are essentially monitoring programs that keep track of statistics relating to parallel execution such as communication bandwidth and latencies. More recent variations, such as Bedy's [13] thread monitoring system and the Gthread [185] visualization package from the PARADE [164] project give somewhat more detailed information about locks and thread status. Getting more detail has typically meant tailoring the visualizer to a particular environment. For example, the Gthread system works only on Kendall Square Research (KSR) machines, Eden [15] is specific to Haskel programs, and Pajè [94] is a visualization system designed for data-flow programs such as those written in the Athapascan [63] environment. Pajè monitors long-lived parallel threads, diagrams blocked states, and illustrates message transfer and latency. Assuming that threads are relatively few and long-lived is typical of many parallel visualizers. In many cases, such as with ParaGraph [73], PARvis [108], and Moviola [105], the visualization as-

sumes one thread per node, and then focuses on the communication and blocked status of those "threads". They provide time-process communication graphs of explicit communication events that make it easy to identify basic communication problems and patterns. Explicit communication is typical of the MPI programming model and MPI visualization tools like Vampir [124] provide similar information in similar-looking graphs. The visualization used in this thesis uses communication behavior to help define structure, rather than presenting a structure based on the hardware layout.

Charting the structure of an application opens the possibility of using that structure to check for programming errors. As multithreaded applications have become more popular, automated correctness checkers have received a great deal of interest. In some cases these tools stem from serial application correctness checkers. This is especially true of memory checkers such as IBM's Rational Purify [163] and Valgrind [127]. Valgrind is particularly interesting because it has developed a validation component, Helgrind [87], to perform validation of common threading operations and watch for potential race conditions. Another similar tools is Intel's Thread Checker [36]. These tools are all dynamic program analysis tools, similar to the tracing tools that generate the event description logs used in this thesis. The approach used here is more akin to shape analysis, such as done by Sagiv et al [150], because of the way it renames memory objects based on access behavior, though this approach relies primarily on thread behavior to inform memory object definition.

2.3 Locality-Aware Data Structures

With a firm understanding of algorithm structures, it becomes possible to design data structures that can bridge between hardware memory layouts and algorithm structures.

2.3.1 Data & Memory Layout

Work in memory layout generally attempts to optimize for one of two things: either cache efficiency or parallelism. Cache efficiency work typically relies on the reuse-distance metric—the number of instructions between uses of a given piece of data—to determine the success of the layout [172, 174], which biases data layout toward the creation of hot-spots [45, 58]. Optimizing data layout for a parallel environment [20, 70, 104], however, relies upon the observation that hot-spots prevent multiple nodes from efficiently accessing data concurrently and incurs large cache coherency overheads [10, 54]. Such problems are exacerbated by the "false-sharing" [53] issue, wherein independent data that shares the smallest resource block managed by the cache (a cache-line) is indistinguishable from shared data and thus parallel access to such independent data incurs cache coherency overhead as if it was shared. Thus, data layout for parallel execution requires a bias away from the creation of hot-spots. For example, Olivier Temam developed a compile-time technique for eliminating cache conflicts (hot-spots) through the use of data copying [173].

In a NUMA shared memory parallel system both goals apply. Because memory latency is lower when accessing local data, it is preferable for each thread to execute as close to the data it manipulates as possible. Within that local memory, accesses to independent memory blocks are fastest when reuse-dis-

tance is minimized. At the same time, shared data must be kept separate so that it can to be accessed in parallel while incurring as little cache coherence overhead as possible. These goals are inherently opposed. The optimal layout must balance the two goals according to the granularity of parallelism available and the amount of parallelism in use during a specific execution. Byoungro So, et al. demonstrated the performance potential of customized data layout algorithms for different applications [162]. Ding, et al. demonstrated that runtime modifications to data layout can provide significant improvements in application performance, particularly when the application's memory behavior changes over time [48]. It seems obvious that integrating custom data layout algorithms with standardized application behavior abstractions would yield significant performance improvements.

Layout techniques for NUMA shared-memory machines are a younger field than general-purpose optimization. In 2001, Jie Tao et al. developed a method for analyzing NUMA layouts in simulation [170] and found that applications that are not tailored for the layout of the specific NUMA machine frequently face severe performance penalties [169]. This expanded upon and confirmed work in 1995 by Chapin, et al. who analyzed UNIX performance on NUMA systems [28]. Abdelrahman, et al., among others, have developed compiler-based techniques for arranging arrays in memory to improve performance on NUMA machines [1], however the techniques are static and in addition to working only for arrays of data or other language-supported data types, they also optimize for specific memory layouts or a specific machine size and a specific level of parallelism.

In shared memory systems, the cost of communication for remote memory references is the single largest factor in application performance [79, 125, 142, 151, 158, 159]. Counterintuitively, locality, in the sense of NUMA memory topology, is generally independent of threading interfaces. This is probably a result of the relatively low variance in memory access latency of previous shared-memory machines. That observation, in 2000, led Nikolopoulos et al. [132] to suggest that explicit data distribution was unnecessary due to the low variance in memory latency across NUMA systems: automatic page-based adaptation to application behavior was deemed sufficient. To understand this situation, it is important to consider the ratio of processor speed to memory speed. Large-scale shared memory systems are considered those with hundreds of nodes [80, 90]. Larger systems are sufficiently complex that they require significantly higher memory latencies, which negatively impacts the ratio of processing speed to memory access speeds. Larger systems that behave like shared memory systems have been implemented as a virtual memory layer over a distributed memory system [59, 106], but such implementations incur particularly large overheads [71] that can be avoided when the shared address space is implemented directly in hardware. In any case, such large shared memory systems have, fundamentally, two different memory access latencies: memory access is either extremely fast, or extremely slow. Because of the extreme penalty associated with any remote data access, a message-passing interface [61] has proven to be a useful way of encouraging programmers to avoid the communication overhead. As a result, the complexity of providing a shared-memory interface to systems of more than a few hundred nodes has been largely unnecessary. Thus, Nikolopoulos et al. were cor-

rect in their observation primarily because machines that do not provide relatively uniform access latencies also tend to avoid providing a shared-memory environment. Modern and future large-scale shared-memory machines have higher latency variances, and the effect of these variations has been magnified by processor performance increases [169].

2.3.2 Locality Information

Because threading and locality have been kept separate, interfaces for capturing topology information tend to be operating-system specific. Linux systems largely rely on the libnuma [96] library to provide locality information and exploit system topology. This library presents the system as a linear set of numbered nodes, CPUs, and "physical" CPUs, which overlap arbitrarily. Threads can bind themselves to any combination of nodes, CPUs and physical CPUs. The relationships between nodes are described by a unitless distance metric presenting latency of remote memory as relative to the latency of local memory. The latency of a node's own memory is normalized to a value of 10, and remote access latencies are expressed on that relativistic scale. Thus, if local-memory latencies are not uniform across a system, these distances cannot be directly compared even between processors. Without libnuma, Linux processes can define their CPU affinity using the `sched_setaffinity()` interface. Unfortunately, this interface changes across kernel versions. Software that must work reliably across multiple Linux systems has to detect the specific variety of the interface that is provided. The Portable Linux Processor Affinity (PLPA) library [171] provides a stable interface for setting processor affinity on Linux, but does not provide a means of quantifying distances between

processors. Solaris systems provide the liblgrp library [167], which provides a hierarchical description of the machine's structure. Each locality group (lgrp) is associated with a specific block of memory and can contain a set of CPUs and/or additional locality groups. Recent versions of the library provide a way of measuring the latency between locality groups in machine-specific unspecified units. All of these interfaces enable the establishment of heavyweight thread CPU affinity and the libnuma and liblgrp interfaces enable location-specific memory allocation. Because these interfaces manipulate the operating system's scheduler and memory subsystem, they can only affect things that the operating system schedules itself, which are inherently heavyweight threads.

2.3.3 Data Structures

Likely due to the difficulty in establishing portable, reliable memory affinity, most existing concurrent shared-memory data structures ignore locality concerns. The concurrent vectors, hashes, and queues provided by Intel's Threading Building Blocks [84] interface are good examples of modern data structures designed for parallel operation in shared-memory computers. They are designed around thread-safe concurrent management operations rather than concurrent data manipulation operations, and thus ignore data locality.

Unlike many other data structures, arrays have been the subject of some research that considers locality. For example, Co-array Fortran [134] provides an intelligent location-aware parallel container — the co-array — that neatly integrates locality with the execution model, albeit using loader-defined static distribution and a customized language. In this work, arrays are distributed at

runtime, similar to Global Arrays [129, 130]. Global Arrays, with their put/get semantics, rely on the Aggregate Remote Copy Interface (ARMCI) [128, 131] as a primary communication layer and as such require a message-passing library such as MPI for distributing data. This gives Global Arrays portability to cluster computers, at the expense of copying data multiple times to encapsulate for communication. The distributed array mechanism presented in Chapter 5, like all of the work in this thesis, is designed for a shared memory system and thus exploits the communication efficiencies of such environments.

Research into queue-like data structures generally focuses either on parallel efficiency or single-producer single-consumer speed. A distributed queue is a special case of a concurrent pool [113], which is designed for parallel efficiency. The distributed queue presented in Chapter 5 combines the designs of the "stochastic distributed queue" by Johnson [91] and the push-based distributed queue by Arpaci-Dusseau et al. [11] with locality-based decision making. Internally, it uses multiple lock-free queues based on the design by Michael and Scott [117], because of its high single-producer single-consumer speed.

2.4 Application Behavior

Estimating the impact of new data structures on high-performance software is extremely difficult because such software is typically tightly integrated with the data structures that it uses, whether they be adaptive blocks [165], sparse matrices [144], distributed arrays [130, 134], or something else. This difficulty is widely recognized as a considerable challenge when developing representative benchmarks. For example, the Mantevo project [76] attempts to model high performance computing applications by developing mini ap-

plications that use data structures similar to those used in high performance software in a way that mimics the way those structures are used in that software.

High performance software for large-scale systems is sometimes written around a computational abstraction rather than a data structure. A computational abstraction can be thought of as a computation "template" that defines the structure of the computation, and with that, the communication patterns. The simplest example of such an abstraction is the "bag-of-tasks" abstraction, where a list of unordered and independent computational goals are defined and executed. This computational template has been successful in a variety of situations, including Linda [4], Condor [109], and Seti@Home [166]. Bulk-Synchronous-Parallel (BSP) is a similar abstraction, designed to allow some dependence and communication between tasks. Even well-understood basic computational tasks, such as sorting, can be considered to be a computational abstraction [49], depending on the situation. Another popular example of a computational abstraction is Map-Reduce [43]. Map-Reduce is far more structured than the BSP or sort abstractions, and has proven particularly efficient for specific categories of problems and data sets. Map-Reduce applies a function to every item in the input set, producing an ordered list of key-value pairs. These pairs are grouped by their keys, and a reduction function is applied to each key and its associated list of values to produce a single set of output values. This abstraction has, like the "bag-of-tasks", proven useful in a variety of situations, most significantly in computing the Google pagerank index of the World Wide Web [25].

This thesis examines two abstractions in particular: All-Pairs and Wavefront. All-Pairs [120] computes the Cartesian product of two sets of data, applying a function to each pair of elements from both sets and storing the result in a lookup table of some type. This abstraction has proven useful in, for example, bioinformatics and data mining applications. The Wavefront [184] abstraction defines a recurrence relation in two dimensions. It establishes a virtual output matrix where each element in the output is a function of its neighbors in the preceding row and column and the input is the first row and column of the matrix. This is useful in a variety of simulations problems in economics and game theory, as well as the dynamic programming sequence alignment problem in genomics.

Computational abstractions and their intrinsic workflows and computational patterns are not applicable to all situations, and are clearly less expressive than even general purpose workflow languages such as DAGMan [175], Dryad [85], Pegasus [44], and Swift [186]. But specialization provides a clear computational structure that can be analyzed and optimized for whatever hardware architecture and topology is available.

CHAPTER 3

QTHREADS: COMBINING LOCALITY AND LIGHTWEIGHT THREADING

3.1 Introduction

The previous chapter reviewed a wide variety of threading models. Large-scale multithreading, especially at teraflop-scale and beyond, has requirements that are somewhat unusual and do not match the features of existing threading interfaces. This leads to the obvious question: what should a threading interface for a teraflop-scale multithreaded system look like? A teraflop-scale multithreading interface will, undoubtedly, provide lightweight threads, lightweight hardware-assisted synchronization, and integrated machine topology information.

A teraflop-scale lightweight threading interface needs to:

- support millions of threads,
- handle machine topology,
- be portable across systems,
- be usable on small systems,
- and support lightweight synchronization.

This chapter presents the qthread application programming interface (API), designed to provide the features necessary for teraflop-scale multithreading. It provide fast access to hardware-accelerated threading and synchronization primitives when available, and emulates them when they are not available. The interface is used to parallelize an example benchmark, HPCCG, and scales well. It is also used to port a specialized high-performance parallel code from the Cray ThreadStorm architecture to commodity systems.

3.2 Background

Several multithreading trends benefit from lightweight threads: large scale threading, latency toleration, and the idea of thread migration. Managing millions of threads and the resources required for them is, in itself, a significant problem. Teraflop-scale multithreading requires massive amounts of parallelism — both hardware and software. At that scale, per-thread overhead becomes a critical issue that makes only lightweight threading models feasible. Multithreading is also often used as a means of tolerating memory latency, which means that multiple threads must be held in processor memory at the same time so that the processor can switch between them quickly. Each additional byte of context required per thread increases the on-chip storage requirements for maintaining multiple hardware threads. The concept of thread migration [40, 88, 122] is an idea gaining traction as a means of exploiting memory topology. However it is only a useful option if migrating a thread is less expensive than the remote accesses that are avoided. Thus, lightweight threads increase the viability of thread migration.

Recent hardware architectural research has investigated lightweight threading and programmer-defined large scale shared-memory parallelism. The lightweight threading concept allows exposure of greater potential parallelism than heavyweight threading or message-passing interfaces, increasing performance by exploiting any additional available fine-grained hardware parallelism. For example, the Sun Niagara [100] and Niagara 2 [92] processors provide support for four and eight concurrent threads per core, respectively, with up to eight cores per processor. Niagara 2 systems support up to four processors, for a total of 256 concurrent threads per system. As a larger example, the Cray XMT [39], with the ThreadStorm CPU architecture, avoids memory dependency stalls by switching between 128 concurrent threads per processor. XMT systems support over 8,000 processors at a time. Thus, to maximize throughput, the programmer must provide at least 128 threads per processor, or over 1,024,000 threads in a full-size system.

Lightweight threading requires a lightweight synchronization model [180]. There are several options, including ADA-like protected records [110] and forksync [16] — both of which lack a clear hardware analog — as well as transactional memory and full/empty bits. Transactional memory is a synchronization model that started in hardware, as the Load-Linked/Store-Conditional feature of many RISC processors, and became a popular implementation design in databases. It has been implemented in both hardware [30] and software [35, 46, 154], but despite a great deal of academic interest [8, 26, 42, 119], has historically proven to incur too much overhead and has too many unresolved semantics issues to be practical for fine-grained general purpose computing [22]. The model used by the Cray MTA/MTA-2/XMT and PIM designs,

pioneered by the Denelcor HEP [51], is full/empty bits (FEBs). This technique marks each word in memory with a "full" or "empty" state. Threads may wait for either state, and atomic operations on a given memory word's contents also atomically affect its status. This technique can be implemented directly in hardware, as it is in the Cray systems.

Large parallel systems provide inherently non-uniform memory access latencies. As the size of parallel systems has grown, the non-uniformity of their memory latencies has increased. This trend is likely to continue. Standard threading interfaces—such as pthreads [83], OpenMP [137], and Intel Threading Building Blocks [84]—are designed for basic symmetric multiprocessing (SMP) and multicore systems. They provide powerful and convenient tools for developing software tailored to such systems. While these interfaces make it easy to create parallel tasks, they incorrectly assume that all data is equally accessible from all threads and so do not maintain data and thread locality nor provide tools to discover and exploit system topology. Some operating systems attempt to address these shortcomings by providing mechanisms both to discover machine topology and to specify the CPU affinity of threads and allocated memory blocks. However, using these mechanisms directly requires using system-specific libraries with non-portable interfaces that only affect heavyweight threads. Lightweight threading models, such as OpenMP, often use locality-ignorant work-stealing scheduling techniques that exacerbate the problem. In order to properly define and schedule threads, both the application and the thread scheduler must be aware of the topology of the system resources in use, and thus unifying threading and locality interfaces is a logical development.

Large scale multithreaded systems each use different topologies and synchronization mechanisms. If they support lightweight threading, the threading interface is typically system-specific as well. Not only does this situation make comparisons between large systems difficult, it also complicates single-system software development. Because lightweight threading primitives are typically either platform-dependent or compiler-dependent, developing multithreaded software for large scale systems often requires direct access to the large system in question. Direct access is not always convenient because of expense, access restrictions, or contention for machine time. Thus, the best threading interface is one that provides the same features and interface on both large and small systems.

This chapter introduces the qthread lightweight threading API and its Unix library-based implementation. While library-based multithreading imposes some semantic issues when optimizing code [17], it is a convenient method for developing and discussing a fundamental and portable threading interface. Such an interface can be used as a programming target by compilers to implement multithreaded applications, thereby avoiding the semantic issues. The API is designed to be a lightweight threading interface that maps well to future large scale multithreaded architectures while still providing performance, portability, and utility on commodity systems. The Unix implementation is a proof of concept that provides a basis for developing applications for future architectures.

3.3 Qthreads

This chapter presents the qthread lightweight threading API, which provides several key features:

- Large scale lightweight multithreading support
- Inherent thread locality
- Basic thread-resource management
- Access to or emulation of lightweight synchronization mechanisms
- Source-level compatibility between platforms
- Threaded loop operations
- Distributed data structures and computational abstractions

The qthread API, the basic aspects of which are summarized in Figure 3.1, was designed to maximize portability to experimental architectures supporting lightweight threads and unusual synchronization primitives while providing a stable interface to the programmer for using these lightweight threads. The qthread API integrates locality control with the threading interface and includes distributed data structures and computational abstractions that make use of that integrated control.

In addition to portability, performance is important to the design of the qthreads API. The goal is to add as little overhead as possible to experimental architectural threading primitives while maintaining a generic interface. The

Threads	Synchronization	Futures
qthread_init(*n***)** Initialize the library with *n* locations **qthread_fork(***f*,*arg*,*ret***)** Create a lightweight thread to execute *ret = f(arg)* **qthread_fork_to(***f*,*a*,*r*,*shep***)** Create a lightweight thread to execute *r = f(a)* on *shep* **qthread_self()** Retrieve a thread's self-reference **qthread_id(***self***)** Retrieve a thread's unique identifier **qthread_shep(***self***)** Discover which shepherd a thread is assigned **qthread_stackleft(***self***)** Discover the remaining stack space **qthread_retloc(***self***)** Discover the location the return value will be stored, if any **qthread_finalize()** Clean up threads and free library memory	**qthread_lock(***self*,*addr***)** Block until the address *addr* is locked **qthread_unlock(***self*,*addr***)** Unlock the address *addr* **qthread_readFF(***self*,*dst*,*src***)** Block until *src* is full, copy a machine word from *src* to *dst* **qthread_readFE(***self*,*dst*,*src***)** Block until *src* is full, copy a machine word from *src* to *dst*, mark *src* empty **qthread_writeF(***self*,*dst*,*src***)** Copy a machine word from *src* to *dst*, mark *dst* full **qthread_writeEF(***self*,*dst*,*src***)** Block until *dst* is empty, copy a machine word from *src* to *dst*, mark *dst* full **qthread_incr(***addr*,*incr***)** Atomically add *incr* to the integer value in *addr* **qthread_dincr(***addr*,*incr***)** Atomically add *incr* to the floating point value in *addr* **qthread_cas(***addr*,*oldv*,*newv***)** Atomically swap *newv* into *addr* if *addr* contains *oldv*	**future_init(***n***)** Specify that each location may have only *n* futures **future_fork(***f*,*arg*,*ret***)** Block until a future is created to execute *ret = f(arg)* **future_fork_to(***f*,*a*,*r*,*shep***)** Block until a future is created to execute *r = f(a)* on *shep* **future_yield(***self***)** Stop counting this thread as a future **future_acquire(***self***)** Block until this thread can be counted as a future

Figure 3.1. The Qthread API

qthread API consists of four components: the core lightweight thread command set, an interface for basic threaded loops, a set of distributed data structures that use the built-in locality information, and a set of computational abstractions. The data structures are discussed in Chapter 5, and the computational abstractions are discussed in Chapter 6.

3.3.1 Semantics

More than simply a threading implementation, though, a qthread is a lightweight threading concept intended to match future hardware threading environments and features more closely than existing threading concepts. How-

ever, given the large body of existing programs with pthreads, qthreads are intended to be similar. A lightweight thread, or "qthread," as defined here, is a nearly-anonymous thread with a small (e.g. one page) stack that executes in a particular location. These three crucial aspects—anonymity, limited resources, and inherent localization—represent significant and fundamental changes in the way explicit threads function.

The anonymity of these threads means that they cannot be controlled by other threads other than through explicit synchronization operations. For example, an existing thread cannot be canceled by another thread. However, the threads are only nearly anonymous because they can be uniquely identified in at least two ways. First, each thread, when run, is given a pointer to an opaque data structure representing itself for speed when doing introspection. Secondly, each thread *can* have a return value location, which can be used to wait for that thread to exit. When the thread is created, this return value location is emptied with the builtin FEB locking mechanism. Then, when the thread exits, the location is written to and marked as full. This form of anonymity is similar to "detached" pthreads in that every thread can clean up all of its resources immediately after it exits, but doesn't assume hardware that handles thread-specific signals.

It is assumed that threads operate in computers of finite size and with limited resources. In systems with finite size that use large numbers of threads, threads must deal with having limited resources. For example, in a 32-bit environment with one million threads, if each thread receives non-overlapping thread-specific state then each thread can have at most four kilobytes of thread-specific state, which would leave no remaining address space for any other

data. In recognition of this, and to deal with the architecture-defined variability in stack storage, a qthread's thread-specific storage (i.e. stack) is both explicitly limited and can be introspected programmatically.

Other threading systems are also naturally limited, but the limits are less obvious. For example, by default, pthread threads typically have relatively large eight megabyte stacks allocated for them, which can be resized dynamically as necessary. Such dynamic resizing requires that thread stacks be allocated with logical space between them. Stack overruns in pthread systems are detected and handled by creating page fault signal handlers and by marking the next page of memory after the end of the stack as unreadable. Thus, no matter the stack size, each thread's stack must be logically separated from other thread stacks by at least a page to support overrun detection, with additional space to allow for resizing. This overhead is implicit in the pthread model, rather than explicit. The pthread interface allows for thread stack sizes to be specified, allowing the programmer to limit the impact of a given thread, but the pthread interface provides no introspective ability to discover the amount of stack space used, thus relegating pthread stack size limits to the "guess-and-check" category of feature where guessing incorrectly leads to either unexpected crashes or memory corruption.

That each thread is inherently localized—that is, executes in a specific location—is quite the opposite of the fundamental assumptions of most other threading models. For example, pthread threads make no guarantees about a thread's location. There are system-specific tools, such as libnuma and liblgrp, that allow thread locations to be specified after the threads have been created, but that is an inefficient model, at best. Thread-specific data is, in such mod-

els, allocated somewhere in the system when threads are created. Then, when a location is specified, it must be moved to the thread's new location. Additionally, the system-specific nature of such locality interfaces inhibits porting location-aware applications between systems.

There is a psychological aspect to the localization characteristics of the interface as well. By integrating locality with the threading interface, the programmer is afforded control over memory locality in a way that makes the balkanization of ccNUMA systems explicit, thereby allowing it to be part of the design process. In any parallel system of sufficient size, there are distributed resources and therefore there is communication between those resources. The communication is usually the single most expensive action the applications take, particularly in large systems. One of the things that has made MPI successful is that it provides a framework that forces programmers to consider communication as something of an inconvenience, which encourages them to keep it to a minimum. Good logical separation between parallel threads is achieved by encouraging a helpful thought process. In multithreaded applications, as opposed to MPI, communication is implicit rather than explicit. Without explicit and unavoidable locality, communication is difficult to think about and can easily happen unintentionally. By making location an obvious, reliable, and unavoidable aspect of every qthread, the qthread concept turns locality into something that is convenient to reason with and take advantage of.

3.3.2 Basic Thread Control and Locality

Qthreads lack most of the expensive guarantees and features of their larger process-like cousins, such as per-thread process identifiers (PIDs), signal vectors, and the ability to remotely cancel threads. This lightweight nature supports very large numbers of threads with fast context-switching between them, enabling high-performance fine-grained thread-level parallelism. The API also provides an alternate form of thread, called a "future", which is created only as resources are available.

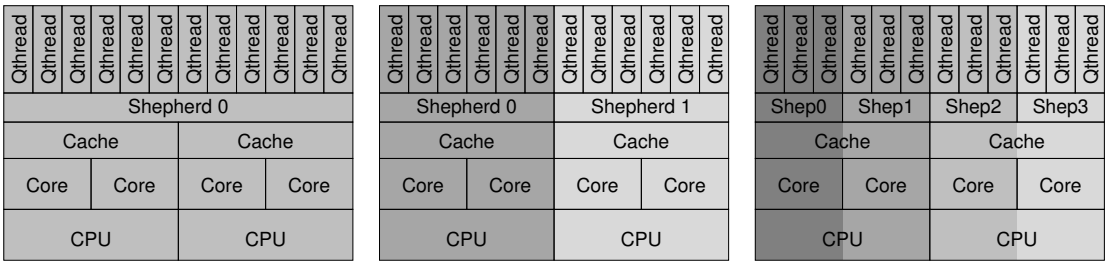
Qthreads and futures, once created, cannot be directly controlled by other threads. However, they can provide FEB-protected return values so that a thread can efficiently wait for another to complete. Among other benefits, this allows qthreads to release their resources when they exit even if no other thread is waiting for them. FEBs do not require polling, which is discouraged as the library does not guarantee preemptive scheduling.

The scheduler in the qthread library uses a cooperative-multitasking approach, though on architectures with hardware support for lightweight thread scheduling, they may be preemptively scheduled. When a thread blocks, such as from a lock or FEB operation, a context switch is triggered. Because these context switches are done in user space via function calls and therefore do not require signals or saving a full set of registers, they are less expensive than operating system or signal-based context switches. This scheduling technique allows threads to take full advantage of the processor until data is needed that is not yet available, and allows the scheduler to hide communication latencies by keeping the processor busy with useful work. Logically, this only hides communication latencies that take longer than a context-switch.

Only two functions are required for creating threads: `qthread_init(shep)`, which initializes the library with `shep` shepherds; and `qthread_fork(func,arg,ret)`, which creates a thread to perform the equivalent of `*ret = func(arg)`. To destroy all threads and release all library-specific memory, use `qthread_finalize()`.

Qthreads are always associated with—and scheduled in the bailiwick of—a thread mobility domain, or "shepherd," which is assigned when the thread is created. These shepherds are grouping constructs that define immovable regions within the system. Shepherds establish a location within the system, and may correspond to nodes in the system, memory regions, or protection domains. The number of shepherds is defined when the library is initialized. In the Unix implementation, a shepherd is managed by at least one pthread which executes qthreads. It is worth noting that this hierarchical thread structure, particular to the Unix implementation (not inherent to the API), is not new but rather useful for mapping threads to mobility domains. Hierarchical threads have also been used by the Cray X-MP [168], as well as Cilk [16] and other threading models.

Figure 3.2 illustrates several ways of mapping shepherds to a dual-processor dual-core machine, and displays the relationship between qthreads and shepherds. Using only two shepherds on this system, as in Figure 3.2(b), establishes one shepherd per cache: each qthread will always execute using the same cache. Using four shepherds, as in Figure 3.2(c), maps each shepherd to a core, ensuring that each qthread will always execute on the same core. Figure 3.2(a) illustrates the use of only a single shepherd, thereby providing no guarantee as to where in the system each qthread will execute. In each figure,



(a) One (b) Two (c) Four

Figure 3.2. The Shepherd/Qthread Relationship

each qthread of a given color can only execute on the shepherd—and thus on the hardware—of the same color.

Qthreads can be spawned directly to a specific shepherd, if desired, via the `qthread_fork_to(func,arg,ret,shep)` function. Qthreads can look up information about themselves; for example `qthread_retloc()` returns the address of the return value of the thread, and `qthread_stackleft()` provides an approximate estimate of the bytes remaining in the thread's stack. The currently assigned shepherd can be queried with `qthread_shep()`. Once assigned a shepherd, qthreads can move between shepherds by calling the `qthread_migrate_to(shep)` function. This movement is always explicit rather than implicit, thereby providing a scheduling guarantee that other threading interfaces do not: a thread cannot execute in an unexpected location. This guarantee allows the programmer greater control over the placement of threads and communication between machine locations. Such control previously required the use of platform-specific libraries. The distance between shepherds may be determined using the `qthread_distance(src,dest)` function. For convenience when making decisions about locality, migration, and

memory distribution, a list of shepherds sorted by distance from the caller can be efficiently obtained via the `qthread_sorted_sheps()` function.

Though the API has no built-in limits on the number of threads, thread creation may fail or need to be restricted due to memory exhaustion or other system-specific limits. Even lightweight threads consume resources to exist, such as the cost of storing their processor context. While the programmer certainly could track these resources and integrate restrictions on threading into the design of an application, it can be useful to restrict the creation of threads — and thus their resource consumption — without reimplementing a resource tracking mechanism for each application. "Futures" provide a generic way to restrict thread creation. Futures are qthreads that are limited to a programmer-defined per-shepherd maximum. Futures are created with `future_fork(func,arg,ret)`, which has semantics similar to those of `qthread_fork()`. The programmer can impose a per-shepherd limit of n futures with the function `future_init(n)`. Thereafter, calls to `future_fork()` that would result in more than n futures existing on any given shepherd will block. When one of the existing futures exits, resulting in fewer than n futures existing, one of the blocked `future_fork()` operations will unblock and succeed.

3.3.3 Synchronization

Access to hardware features, such as atomic operations, is critical for optimizing performance on any given computer. Virtually all computer architectures provide atomic hardware primitives, such as an increment or a compare-and-swap. However, atomic operations are not a standard part of most programming interfaces. Even basic operations that are frequently atomic, such

as single-word reads or writes, have different memory semantics on different systems.

The complexity of portable atomic operations is easy to demonstrate. For example, a hardware-accelerated atomic increment on a Xeon processor is a hardware primitive, and using it is fairly straightforward. The Xeon uses the AMD64 ISA, which simply requires an `xadd` instruction with the `lock` prefix, along with sufficient compiler directives to prevent the value of the given memory address from being cached across the increment. The Niagara 2 processor, using the SparcV9 ISA, provides only an atomic compare-and-swap hardware primitive. Thus, atomic increments on a Niagara 2 must be performed in a loop using the `cas` instruction with a `membar` to assure memory dependencies are respected. Doing the same atomic increment on an Itanium processor, which uses the IA64 architecture, requires both approaches. In many cases, a `fetchadd` instruction can be used, similar to AMD64's locked `xadd`. However, the `fetchadd` instruction can only work with a limited set of increment values. For arbitrary increment values, the increment must be done using a compare-and-swap loop: atomically read the current value, perform the addition, copy the old value into the comparison criterion application register (`ar.ccv`), and then use a compare-and-swap instruction (`cmpxchg`) to either place the incremented value into the input address or restart the loop. The PowerPC architecture provides transaction-like load-locked/store-conditional instructions. Thus, an atomic increment on a PowerPC must also be performed in a loop. The value to be incremented must first be loaded and its address reserved with the `lwarx` instruction. Then the value is incremented and the new value stored back to the original address with the conditional

`stwcx.` store instruction, which will fail if the reservation no longer exists. If the store fails, the loop must be restarted. If the store succeeds, an `isync` instruction must be issued to enforce memory dependencies. Of course, these operations change subtly depending on the bitwidth of the incremented value.

The qthread library provides cross-platform access to atomic increments using a simple `qthread_incr(addr,val)` inline function call. A similar nest of complex assembly instructions is necessary to provide an atomic compare-and-swap operation across multiple platforms, which the qthread library provides via the `qthread_cas(addr,old,new)` inline function call. It is often convenient in scientific computing to have atomic floating-point increments as well, and the qthread library provides both `qthread_fincr(addr,val)` and `qthread_dincr(addr,val)` to increment floating point values atomically.

These non-blocking atomic operations complement the blocking synchronization methods that the qthread library provides: basic mutex-like locking and full/empty bit (FEB) operations. The mutex operations are `qthread_lock(addr)` and `qthread_unlock(addr)`. The FEB semantics, which allow memory reads and writes to block based on a per-word full or empty status, are more complex. They include functions to manipulate the FEB state in a non-blocking way (`qthread_empty(addr)` and `qthread_fill(addr)`), as well as blocking reads and writes. The blocking read functions wait for a given address to be full and then copy the contents of that address elsewhere. One (`qthread_readFF()`) will leave the address marked full, the other (`qthread_readFE()`) will then mark the address empty. There are also two write actions. Both will fill the address being written, but one (`qthread_writeEF()`) will wait for the address to be empty first, while the other (`qthread_writeF()`)

won't. Using both mutex and FEB operations on the same addresses at the same time produces undefined behavior, as they may be implemented using the same underlying mechanism.

Because blocking synchronization affects thread scheduling and can trigger context switches, it is inherently more complex than atomic operations, but provides a powerful way to express dependencies between threads. While some systems, such as Cray MTA and XMT systems, provide hardware FEB support, on most architectures the qthread library must emulate them. Using blocking synchronization methods external to the qthread library is discouraged, as they would prevent the library from making scheduling decisions.

3.3.4 Threaded Loops and Utility Functions

The qthread API includes several threaded loop interfaces, built on the core threading components. Both C++-based templated loops and C-based loops are provided. Several utility functions are also included as examples. These utility functions are relatively simple, such as summing all numbers in an array or finding the maximum value of that array.

There are two basic parallel loop behaviors: one spawns a separate thread for each iteration of the loop, and the other uses an equal distribution technique that spawns a single thread for each shepherd and distributes the iteration space over those threads. Functions that provide one thread per iteration are `qt_loop()` and `qt_loop_future()`, using either qthreads or futures, respectively. Functions that use equal distribution are `qt_loop_balance()` and `qt_loop_balance_future()`. A variant of these, `qt_loopaccum_balance()`, allows

```
unsigned int i;
for (i = start; i < stop; i += stride) {
    func(NULL, argptr);
}
```

(a) C Loop

```
qt_loop(start, stop, stride, func, argptr);
```

(b) Threaded Equivalent

Figure 3.3. `qt_loop()` and C Equivalent

```
void func(qthread_t *me, const size_t startat, const size_t stopat,
          void *arg)
{
    for (size_t i = startat; i < stopat; i++)
        /* do work */
}
```

Figure 3.4. `qt_loop_balance()` User Function Example

iterations to return a value that is collected, or "accumulated", via a user-defined accumulation function.

The `qt_loop()` and `qt_loop_future()` functions take arguments `start`, `stop`, `stride`, `func`, and `argptr`. The functions provide a threaded version of the loop in Figure 3.3(a).

The `qt_loop_balance()` functions, since they distribute the iteration space, require a function that takes its iteration space as an argument. Thus, while similar to `qt_loop()`, they expect that the `func` argument points to a function structured like the one in Figure 3.4.

```
unsigned int i;
void *tmp = malloc(size);
for (i = start; i < stop; i++) {
    func(NULL, i, i+1, argptr, tmp);
    accumulate(retval, tmp);
}
free(tmp);
```

(a) C Loop

```
qt_loopaccum_balance(start, stop, size,
                     retval, func, argptr,
                     accumulate);
```

(b) Threaded Equivalent

Figure 3.5. `qt_loopaccum_balance()` and C Equivalent

The `qt_loopaccum_balance()` functions use an accumulation function to gather return values. The function provides a threaded loop construct similar to the loop in Figure 3.5(a). The `func()` function in this loop is similar to the one in Figure 3.4.

Like the `qt_loop_balance()` function, the accumulator variant uses the equal distribution parallel loop technique. Thus, the `func()` function used is expected to be similar in structure to the one in Figure 3.4, but with one significant difference: the function must store its return value in the memory addressed by `tmp`, which is then passed to the user-provided `accumulate()` function to gather and store in `retval`.

The qthread API also provides utility functions for performing simple tasks. For example, the `qt_double_min()`, `qt_double_max()`, `qt_double_prod()`, and `qt_double_sum()` functions find the minimum, maximum, product and sum, respectively, of all entries in an array of double-precision numbers using

the `qt_loop()` structure. A similar family of functions, `qutil_double_min()`, `qutil_double_max()`, `qutil_double_prod()`, and `qutil_double_sum()`, perform the same tasks using a lagging-loop structure. Similar functions are provided for signed and unsigned integers.

Assuming that each shepherd maps to a single execution context, the balanced loop design allows the computational power of a parallel system to be maximized with a minimum amount of overhead, but presumes that the iterations do not interact or depend on each other in any way, and does not compensate for additional threads that may also be executing.

A lagging-loop design spawns threads for fixed-size segments of the iteration space. The number of threads active at any given time is thus either limited only by the problem size, or limited by the user-defined restriction on the number of active futures. This technique cooperates with unrelated executing threads, and handles interaction between threads better than the equal distribution technique because more than one thread is typically assigned to each shepherd. Thus, if one thread blocks, other threads can execute. Of course, additional threads means more thread management overhead and more context swapping. This loop design also requires a well-chosen segment size, to minimize thread overhead without reducing the parallelism to a point where it cannot cooperate with other unrelated threads sufficiently.

3.4 Performance

The design of the qthread API is based around two primary goals: efficiency in handling large numbers of threads and portability to large-scale multithreaded architectures. The implementation discussed in this section is the

Unix implementation, which is for POSIX-compatible Unix-like systems running on traditional CPUs that do not have hardware support for full/empty bits or lightweight threading, such as PowerPC, SparcV9, IA32, IA64, and AMD64 architectures.

3.4.1 Implementation Details

The Unix implementation of the qthread library uses a hierarchical threading architecture with pthread-based shepherds to allow multiple threads to run in parallel. Lightweight threads are created in user space as a processor context and a small (one page) stack. These lightweight threads are executed by the pthreads. Context-switching between qthreads is performed at synchronization boundaries rather than on an interrupt basis, to reduce the amount of work and state necessary to swap threads. For performance, memory is pooled in shepherd-specific structures, allowing shepherds to operate independently. Much of the qthread library is implemented with lock-free algorithms, including runnable thread queueing, memory pools, and threaded loop tracking.

On systems without hardware FEB support, FEB operations are emulated via a central striped hash table. This table is a bottleneck that would not exist on a system with hardware lightweight synchronization support. However, the FEB semantics still allow applications to exploit asynchrony even when using a centralized implementation of those semantics.

3.4.2 Micro-benchmarks

To demonstrate qthread's advantages, six micro-benchmarks were designed and tested using both pthreads and qthreads. The algorithms in both implementations are identical, with the exception that one uses qthreads as the basic unit of threading and the other uses pthreads. The benchmarks are as follows:

1. Ten threads atomically increment a shared counter one million times each
2. 1,000 threads lock and unlock a shared mutex ten thousand times each
3. Ten threads lock and unlock 1 million mutexes
4. Ten threads spinlock and unlock ten mutexes 100 times
5. Create and execute 1 million threads in blocks of 200 with at most 400 concurrently executing threads
6. Create and execute 1 million concurrent threads

Figure 3.6 illustrates the difference between using qthreads and pthreads on a 1.3GHz dual-processor PowerPC G5 with 2GB of RAM. Figure 3.7 illustrates the same on a 48-CPU 1.5GHz Itanium Altix with 64GB of RAM. Both systems used the Native POSIX Thread Library [50] implementation of Pthreads on Linux. The bars in each chart in Figure 3.6 are, from left to right, the pthread implementation, the qthread implementation with a single shepherd, with two shepherds, and with four shepherds. The bars in each chart in Figure 3.7 are, from left to right, the pthread implementation, the qthread implementation with a single shepherd, with 16 shepherds, with 48 shepherds, and with

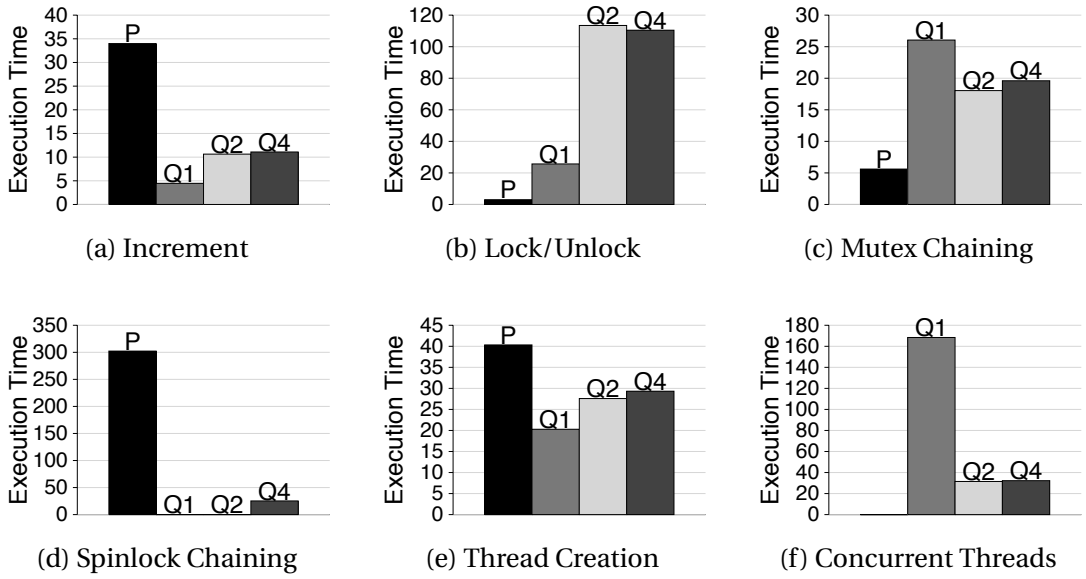
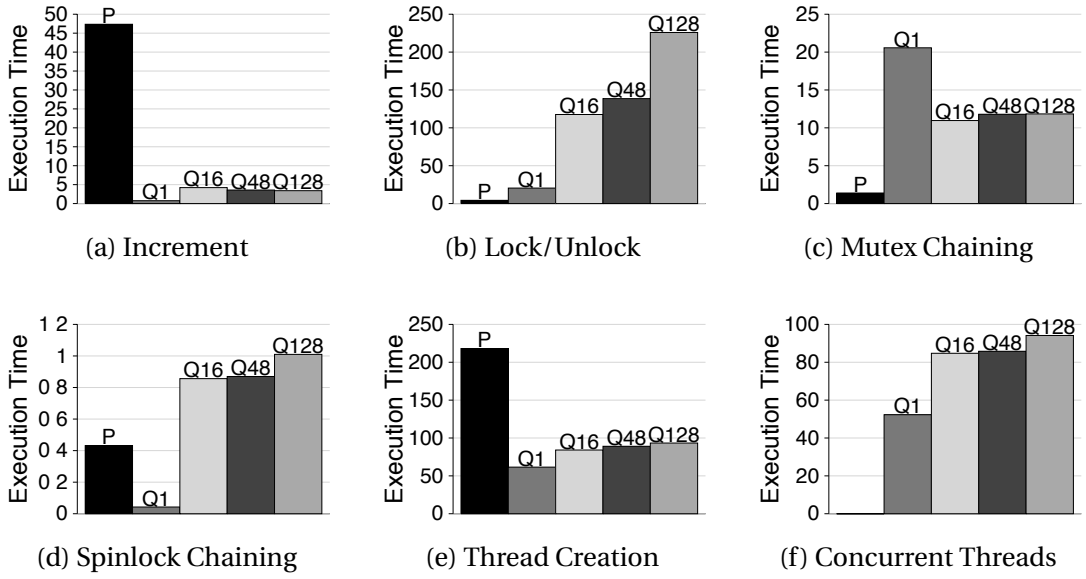


Figure 3.6. Microbenchmarks on a dual PPC

128 shepherds. The way to view these graphs is as a contrast between emulated operations (b, c, and e) and hardware assisted operations (a, d, and f).

Emulation of hardware features is quite expensive. For example, the qthread locking implementation is built with pthread mutexes, and thus cannot compete with raw pthread mutexes for speed, as illustrated in Figures 3.6(b), 3.7(b), 3.6(c), and 3.7(c). This is a detail that would likely not be true on a system that had hardware support for FEBs, and could be improved with a more distributed or efficient data structure, such as a lock-free hash table. Despite the mutex overhead, because of the qthread library's simple scheduler, it is able to outperform pthreads when using spinlocks and a low number of shepherds, as illustrated in Figure 3.6(d). However, the scheduler cannot overcome the costs of mutexes and contention with larger numbers of shepherds (Figure 3.7(d)).



(a) Increment (b) Lock/Unlock (c) Mutex Chaining

(d) Spinlock Chaining (e) Thread Creation (f) Concurrent Threads

Figure 3.7. Microbenchmarks on a 48-node Altix

When the qthread library is able to use hardware primitives, it provides a marked improvement over the pthread alternative. In Figures 3.6(a) and 3.7(a), the pthread implementation is outperformed by the qthread implementation because the qthread library uses a hardware-assisted atomic increment while pthreads is forced to rely on a mutex. Additional shepherds do not improve the qthread implementation's performance because the shared counter quickly becomes a bottleneck. The contention for the counter, and the cache coherency overhead that contention triggers, degrades performance.

While testing the million thread benchmark, in Figures 3.6(f) and 3.7(f), the pthread library proved incapable of more than several hundred concurrent threads—requesting too many threads deadlocked the operating system. Another benchmark was designed to work within pthreads' limitations by using

a maximum of 400 concurrent threads. Threads are spawned in blocks of 200, and after each block, threads are joined until there are only 200 outstanding before spawning a new block of 200 threads. In this benchmark, Figures 3.6(e) and 3.7(e), pthreads performs much better—on the PowerPC system, it is only a factor of two more expensive than qthreads.

3.5 High Performance Computing Conjugate Gradient Benchmark

To evaluate the performance potential of the API and the potential ease of integrating it into existing code, a representative application was modified to use the qthread library. The High Performance Computing Conjugate Gradient (HPCCG) benchmark from the Mantevo project [76] was parallelized with the threaded loops component of the qthread library. The HPCCG program is a C++-based conjugate gradient benchmarking code for a 3-D chimney domain, largely based on code in the Trilinos [77] solver package. The code relies primarily upon tight loops where every loop iteration is essentially independent of every other iteration. With simple modifications to the code structure, the serial implementation of HPCCG was transformed into multithreaded code.

3.5.1 Code Modifications

One of the primary phases of the benchmark is the WAXPBY phase, originally implemented as the relatively simple tight loop in Figure 3.8.

To parallelize these loops with qthreads requires the definition of a structure to pass the arguments to the worker threads, and a worker thread function. A simple structure for passing the arguments is defined in Figure 3.9.

```
int waxpby (const int n, const double alpha, const double * const x,
            const double beta, const double * const y, double * const w)
{
  if (alpha==1.0)
    for (int i=0; i<n; i++) w[i] = x[i] + beta * y[i];
  else if (beta==1.0)
    for (int i=0; i<n; i++) w[i] = alpha * x[i] + y[i];
  else
    for (int i=0; i<n; i++) w[i] = alpha * x[i] + beta * y[i];
  return(0);
}
```

Figure 3.8. HPCCG's WAXPBY Phase

```
struct four_args {
  const double * const y, * const x, beta, alpha;
  double * const w;
  four_args(const double * const X, const double BETA,
            const double ALPHA, const double * const Y,
            double * const W) :
    x(X), beta(BETA), alpha(ALPHA), y(Y), w(W) {}
};
```

Figure 3.9. Structure for Passing WAXPBY Arguments

The worker thread function can be defined as in Figure 3.10.

Finally, the original `waxpby()` function can be modified to use these new functions in a threaded fashion, as in Figure 3.11.

Keep in mind that this parallelization effort is a relatively naïve one. No effort at partitioning data or limiting communication between parallel nodes or avoiding false sharing has been made. This experiment is intended to demonstrate only the effectiveness of applying the qthread library's threaded loop constructs to a more complicated tight loop.

```
static void waxpby_worker(qthread_t *me, const size_t startat,
                          const size_t stopat, void * arg)
{
  const struct four_args *a((struct four_args *)arg);
  const double alpha(a->alpha), beta(a->beta), *const x(a->x), *const y(a->y);
  double *const w(a->w);
  if (alpha==1.0)
    for (int i=startat; i<stopat; i++) w[i] = x[i] + beta * y[i];
  else if (beta==1.0)
    for (int i=startat; i<stopat; i++) w[i] = alpha * x[i] + y[i];
  else
    for (int i=startat; i<stopat; i++) w[i] = alpha * x[i] + beta * y[i];
}
```

Figure 3.10. Worker Function for Threading WAXPBY

```
int waxpby (const int n, const double alpha, const double * const x,
            const double beta, const double * const y, double * const w)
{
  struct four_args args(x,beta,alpha,y,w);
  qt_loop_balance(0, n, waxpby_worker, &args);
  return(0);
}
```

Figure 3.11. Threaded WAXPBY

3.5.2 Results

Figure 3.12 illustrates the benefit of even simple parallelization efforts on a regular problem like the HPCCG benchmark. In this case, the benchmark was run on a uniform 75×75×1024 domain—using approximately 4 GB of memory—on a 48-node SGI Altix SMP. Each data point represents the average execution time of 100 benchmark runs using that number of processes or shepherds. The SGI MPI results are presented to 48 processes, or one process per

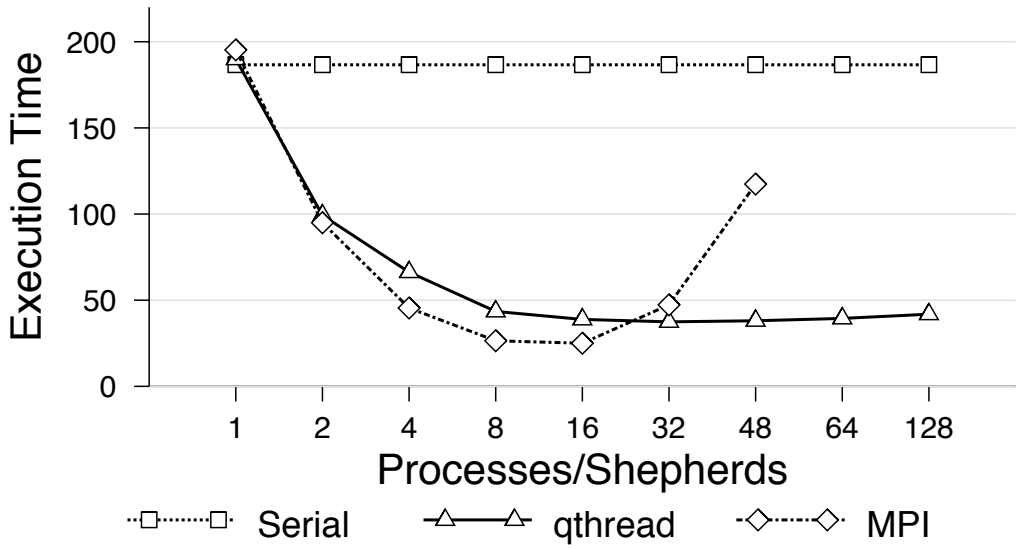


Figure 3.12. HPCCG on a 48-CPU SGI Altix SMP

CPU. Additional MPI processes would over-subscribe the processors, which is a setup that generally underperforms with SGI MPI.

One of the features of the HPCCG benchmark is that it comes with an optimized MPI implementation. The MPI implementation, using SGI's MPI library, is entirely orthogonal to the qthread implementation and does not use threads of any kind. The qthread and MPI implementations scale similarly up to about sixteen nodes. Beyond sixteen nodes however, MPI begins to behave very badly.

The poor performance of the MPI implementation is caused by `MPI_Allreduce()` in one of the main functions of the code. While this takes just under 18.9% of execution time with eight MPI processes, it takes 84.1% of the execution time with 48 MPI processes. While it is tempting to

simply blame the problem on a bad implementation of `MPI_Allreduce()`, it is probably more valid to examine the difference between the qthread and MPI implementations. The qthread implementation performs the same computation as the `MPI_Allreduce()`, but rather than require all nodes to come to the same point before the reduction can be computed and distributed, the computation is performed as the component data becomes available from the threads returning, the computational threads can exit, and other threads scheduled on the shepherds can proceed. The qthread implementation exposes and exploits the asynchronous nature of the benchmark, while the MPI implementation does not. This asynchrony is revealed even though the Unix implementation of the qthread library relies upon centralized synchronization, and thus would likely provide further improvement on a real massively parallel architecture.

3.6 Multi-Threaded Graph Library Benchmarks

Cray's ThreadStorm architecture [38, 39] is of particular interest to lightweight threading researchers, as it provides lightweight threads, hardware FEB support, and a toolchain to take advantage of them. The architecture also has high-performance software available for it. In particular, the Multi-Threaded Graph Library (MTGL) [14] provides a good example of high-performance code written for an architecture that provides these capabilities.

The MTGL is a graph library designed in the spirit of the Boost Graph Library (BGL) [156] and Parallel Boost Graph Library (PBGL) [67]. The library utilizes the generic component features of the C++ language to allow flexibility in graph structures, without changes to the algorithm in use. Unlike the distrib-

uted memory, message-passing-based PBGL, the MTGL was designed specifically for the shared-memory multithreaded ThreadStorm architecture. The MTGL includes a number of common graph algorithms, including breadth-first search, connected components, and PageRank algorithms.

3.6.1 Qthread Implementation of ThreadStorm Intrinsics

The ThreadStorm architecture has several platform-specific features—including FEBs, fast atomic increments, and conditionally created threads—which the qthread API provides. On a ThreadStorm, these features are ordinarily accessed through the use of C-language pragmas that instruct the custom compiler how to parallelize the code. Because ThreadStorm-based applications are so dependent on the compiler, porting such code to other architectures while maintaining its parallel nature is rather difficult.

Conditionally created threads are called "futures" in ThreadStorm architecture terminology, and are used to indicate that threads need not be created now, but merely whenever there are resources available for them. This can be crucial, as each ThreadStorm processor can handle at most 128 threads, while extremely parallel algorithms may naturally generate significantly more. The qthread API's "futures" feature is analogous to the ThreadStorm feature. A key application of this is in loops. While a given loop may have a large number of entirely independent iterations, it is typically unwise to spawn all of the iterations as threads, because each thread has a context and eventually the machine will run out of memory to hold all the thread contexts. Limiting the number of concurrently extant threads limits the amount of overhead that will be used by the threads. In a loop, the option to stall the thread creation

while the maximum number of threads still exist provides the ability to specify a threaded loop without the risk of using an excessive amount of memory for thread contexts and without customizing the loop implementation for any specific machine.

Not all of the ThreadStorm's intrinsic features can be fully emulated. For example, on a ThreadStorm machine, all writes to memory implicitly mark the corresponding memory words as full. Of course, when pieces of memory are being used for synchronization purposes, even implicit operations are done purposefully by the programmer, so replacing implicit writes with explicit calls is trivial. ThreadStorm machines also use an unusual cacheless hashed memory architecture that presents unique performance characteristics: clustered memory accesses that provide performance advantages on commodity systems are a performance hurdle that must be avoided on a ThreadStorm system. Additionally, the Cray compiler for ThreadStorm systems recognizes common programming patterns, such as reductions, and transparently optimizes them. Similarly, it recognizes shared data intrinsically, and can transform some shared data manipulation into atomic operations transparently. For these reasons, developers for the ThreadStorm systems are encouraged to develop "close to the compiler," a habit which slows porting efforts.

3.6.2 Graph Algorithms and Performance

Three different representative graph kernel algorithms in the MTGL were ported to use the qthread interface: a search, a component finding algorithm, and an algebraic algorithm. These algorithms are compared by running them on three different platforms: a Cray XMT, a Sun Niagara 2, and a traditional

multi-core system. The XMT system contains 64 500 MHz ThreadStorm processors and 500 GB of shared memory. Its SeaStar based network is a 3-D torus in an 8×4×2 configuration. The system was running version 6.2.1 of the XMT operating system. The Sun Niagara 2 system is a Sun SPARC Enterprise T5240 server with two 1.2 GHz UltraSPARC T2 processors and 128 GB of FB-DIMM memory. The system was running Sun Solaris 10, 5/08 Release with the Sun Studio 12 CoolThreads version of GCC. The more traditional system is a quad-socket, quad-core 2.2 GHz Opteron with 32 GB of DDR2 memory. It was running RedHat Enterprise Linux 5.1 with GCC 4.1.2.

The algorithms are tested by running them on R-MAT [27] generated graphs with 2^{21}, 2^{23}, and 2^{25} vertices. Two classes of R-MAT graphs are used: "nice" and "nasty." The "nice" graphs feature two natural communities in opposing quadrants with many levels of recursion. The maximum vertex degree in these graphs is very low—in a graph with a quarter of a billion edges, the maximum degree is roughly a thousand. The "nasty" graphs feature a steeper degree distribution, with a maximum degree of $200,000$ in graphs with a quarter billion edges. The idea is that load balancing should be more difficult in the nasty graphs.

3.6.2.1 Breadth-First Search

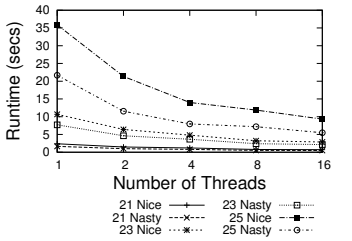
Breadth-first search (BFS) is one of the most fundamental graph algorithms, as it is the basis for developing many other algorithms. Furthermore, it is well-suited for parallelization. The standard BFS algorithm is presented in Figure 3.13. There are two aspects to this algorithm that form bottlenecks in a parallel environment. First, the **for** loop on line 11 will make many synchro-

```
 1: procedure BFS(G, s)
 2:     for all vertex u ∈ V[G] − {s} do
 3:         color[u] ← WHITE
 4:         d[u] ← inf
 5:     end for
 6:     color[s] ← GRAY
 7:     d[s] ← 0
 8:     Q ← ∅
 9:     while Q ≠ 0 do
10:         u ← DEQUEUE(Q)
11:         for all vertex v ∈ Adj[u] do
12:             if color[v] = WHITE then
13:                 color[v] ← GRAY
14:                 d[v] ← d[u] + 1
15:                 ENQUEUE(Q, v)
16:             end if
17:         end for
18:         color[v] ← BLACK
19:     end while
20: end procedure
```

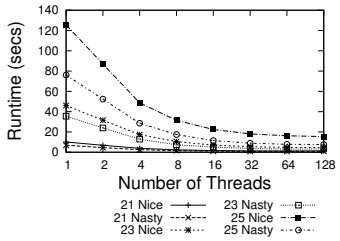
Figure 3.13. The Basic BFS Algorithm

nized writes to the *color* array, forming a bottleneck. The second bottleneck is the ENQUEUE operation of line 15, which typically involves incrementing the queue's tail pointer. The MTGL avoids these problems by chunking and sorting the vertices and by maintaining thread-local copies of the *color* array that are periodically merged.

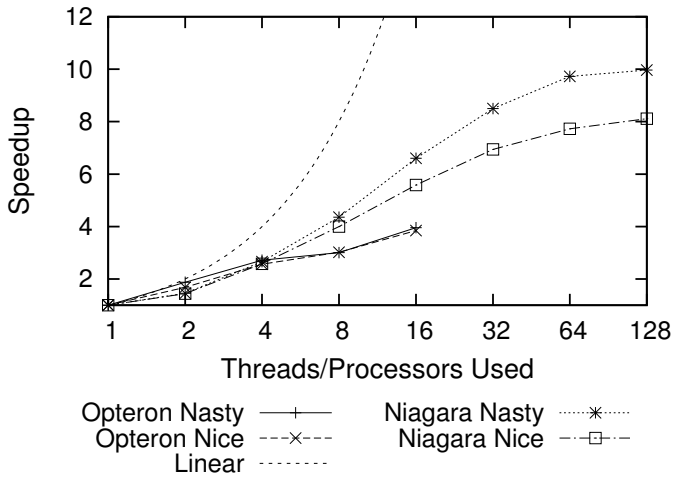
Figure 3.14 presents the results of running the MTGL's BFS algorithm on both the Opteron-based system and the Niagara 2 system. The qthread-based implementation of the BFS algorithm seems to scale effectively on both platforms.



(a) Opteron



(b) Niagara 2

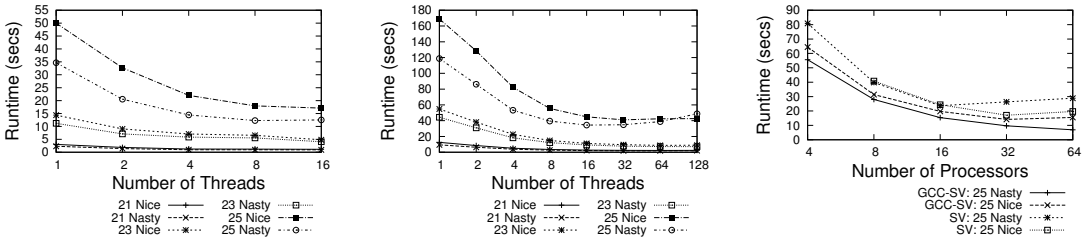


(c) Scaling 2^{25} Graphs

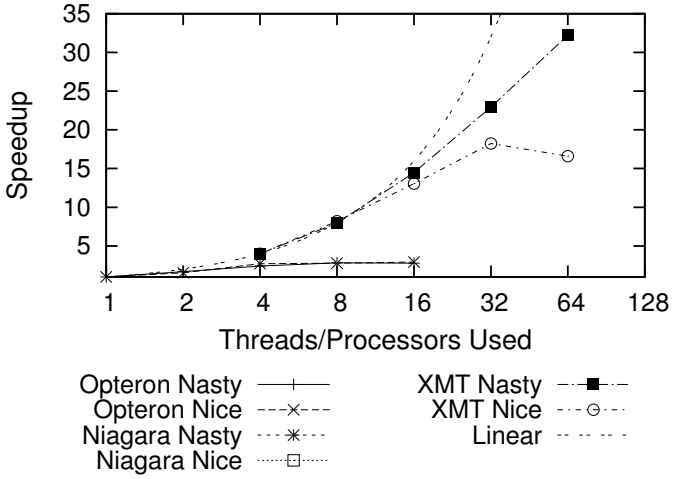
Figure 3.14. Breadth-First Search

3.6.2.2 Connected Components

A connected component of a graph G is a set S of vertices with the property that any pair of vertices $u, v \in S$ are connected by a path. Finding connected components is a prerequisite for dividing many graph problems into smaller parts, and so is particularly important to do quickly. The canonical parallel connected-component algorithm is the Shiloach-Vishkin [155] algorithm (SV). However, both synthetic random graphs [57] and many real-world graphs (such as social networks and the World-Wide Web) have a single "giant



(a) Opteron (b) Niagara 2 (c) XMT



(d) Scaling 2^{25} Graphs

Figure 3.15. Connected Components

component" (GCC). The SV algorithm works by assigning a representative to each vertex. Toward the end of the algorithm, all vertices in the giant component are pointing at the same representative, forming a severe bottleneck. The MTGL includes an alternate algorithm, dubbed "GCC-SV", that leverages the existence of the GCC to optimize the SV algorithm. Figure 3.15 presents the results of running this benchmark with the qthread-based implementation.

The XMT implementation does not use qthreads, and is presented for comparison. The qthread-based implementations, on the Opteron and Niagara 2

systems, do scale. Note that the "nasty" datasets are actually the most friendly to the algorithm, as most vertices fall into the giant component and thus reduce the work the algorithm needs to do. Interestingly, both of the qthread-based implementations top out at approximately four times faster. This is likely due to an unresolved bottleneck in one of the MTGL data structures.

3.6.2.3 PageRank

The PageRank algorithm, a linear algebraic technique for modeling the propagation of votes through a directed graph where each vertex contributes a fraction of its vote to each of its out-neighbors, was made famous by Google for ranking web pages [139]. Ranks continue propagating until the vote totals converge. Figure 3.16 shows the vote accumulation loops of PageRank used by the MTGL on the XMT. The structure of these loops enables the XMT compiler to merge them into one, and to remove the reduction of votes into the variable `total` from the final line of the inner loop. The result is excellent performance. This is simulated in Qthreads, using the explicit `qt_loopaccum_balance()` accumulation loop function, to achieve good scaling on multi-core machines.

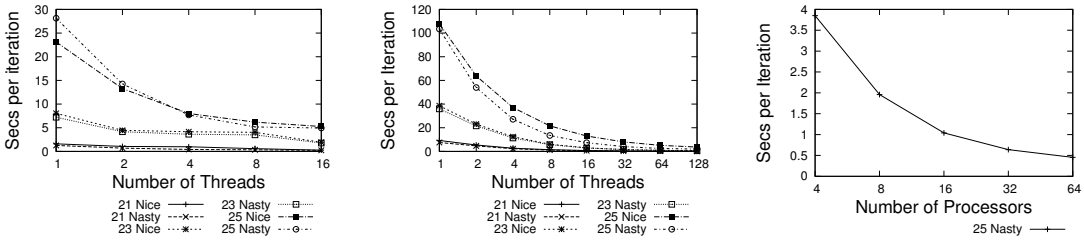
Figure 3.17 presents the results of running the PageRank algorithm. As in the previous benchmark, the XMT implementation does not use qthreads but is included for comparison. While the qthread implementation does not exactly replicate the work of the XMT compiler, it still provides a significant parallel performance benefit.

```
#pragma mta assert nodep
for (int i=0; i<n; i++) {
    double total = 0.0;
    int begin = g[i];
    int end = g[i+1];
    for (int j=begin; j<end; j++) {
        int src = rev_end_points[j];
        double r = rinfo[src].rank;
        double incr = r / rinfo[src].degree;
        total += incr;
    }
    rinfo[i].acc = total;
}
```

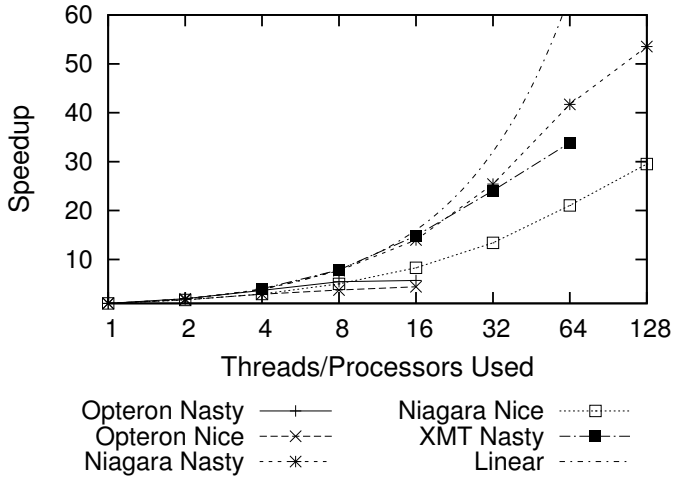
Figure 3.16. Inner Loop of PageRank in the MTGL on the XMT

3.7 Conclusions

Large scale computation of the sort performed by scientific computing computational libraries can benefit significantly from low-cost threading, as demonstrated here. Lightweight threading with hardware support is a developing area of research that the qthread API assists in exploring while simultaneously providing a solid platform for relatively lighter-weight threading on common operating systems. It provides basic lightweight thread control, locality control, and synchronization primitives in a way that is portable to existing highly parallel architectures as well as to future and potential architectures. Because the API can provide scalable performance on existing platforms, it allows study and modeling of the behavior of large scale parallel scientific applications for the purposes of developing and refining such parallel architectures.



(a) Opteron (b) Niagara 2 (c) XMT



(d) Scaling 2^{25} Graphs

Figure 3.17. PageRank

CHAPTER 4

VISUALIZING APPLICATION STRUCTURE WITH THREADSCOPE

4.1 Introduction

The previous chapter discussed some of the issues involved in large scale lightweight threading and presented the qthread lightweight threading API, designed to tackle those problems. However, a threading interface by itself, no matter how intelligent, cannot guarantee performance for a parallel application.

Because of its inherent nondeterminism, multithreaded programming has always been a challenging task. In addition to the standard programming issues, programmers must also avoid errors specific to parallel execution such as race conditions and deadlocks. Synchronization bugs can hide in programs for years, undetected until a specific machine configuration or scheduling order is used. Programmers must also deal with new performance issues like data structure contention and variable levels of available parallelism. Parallelism problems become more complex and hard to analyze or predict as the scale of parallel execution increases, particularly into the teraflop computing range and beyond.

ThreadScope, presented in this chapter, assists the programmer with understanding, troubleshooting, and debugging large scale multithreaded pro-

grams. It is a structural visualization mechanism and visual language for understanding multithreaded programs. ThreadScope uses existing tracing tools to instrument multithreaded applications and uses those traces to visualize the logical structure. The logical structure of multithreaded programs does not rely on a specific order of execution other than that specified by synchronization methods. In many situations, this approach can detect threading problems without needing to replicate them in execution.

The high level structure of a program reveals the program's parallel and sequential components, as well as its potential bottlenecks. This structure is usually independent of the underlying machine, though may be dependent on the program's input. Graphs of the structure can be dense and detail-heavy. The challenge in any dense visualization is deciding where to expand and condense details. To clarify and simplify the visual depiction of the program structure, ThreadScope employs a static single assignment form to remove programming idioms and coalesce memory cells into logical memory objects. These simplification techniques demonstrate how application-specific data structures can be handled. Race conditions and deadlocks are identified by analyzing the graph structure. Unlike some correctness checkers [81], this approach does not require the software to be described in a new special-purpose language, but can address existing applications without modification. A few key operations — reading and writing to memory, synchronization operations, and spawning or joining threads — are the basic structural elements of any multithreaded application. Thus, this analysis technique can be used both in the debugging process and the design process.

This approach relies on fundamental building blocks of multithreaded applications and is not specific to a particular threading library or model. This chapter demonstrates ThreadScope's visualization capabilities and analysis features on several programs using a variety of parallel programming models, and discusses the use of this visualization for structural identification of problems. The programming models used include the Cilk threading library [16], the standard pthread library [136], and the qthread threading library [182]. These models were chosen for their variety of synchronization mechanisms and their explicit parallelism—and the qthread library can generate ThreadScope traces itself—however the ThreadScope technique may be adapted for use with additional environments.

The key advance of this chapter is the ThreadScope visualization tool, which provides a graph-based approach to understanding application structure and identifying errors. It can be used to understand the structure of multithreaded applications for the purpose of bridging the gap between logical and actual parallelism. As discussed in Chapter 2, the key to obtaining portable performance is efficient mapping of logical parallel structure onto the physical machine topology. ThreadScope diagrams are also useful for identify race conditions and deadlocks as well as predict potential bottlenecks and recognize the underlying programming model. Identification of livelocks and analysis of message-passing parallel applications are venues for future work.

4.2 Methodology

The graphs presented in this chapter are the result of a two-stage data collection and analysis process. In the first stage, a program, such as the example

Cilk program in Figure 4.1(a), is traced by an existing tracing tool. This trace is translated into an "event description" language. Tracing the program in Figure 4.1(a) produces the event description in Figure 4.1(b). ThreadScope includes tools for generating event descriptions from the output of several tracing tools, including Dtrace [24], Apple's libamber [6], the SST simulator [148], and the qthread library. Other tracing tools could be used to produce similar event descriptions; the basic requirement is the ability to detect thread and synchronization operations. In the second stage, ThreadScope uses the event description to generate dot attributed graph language, which is rendered by the GraphViz [64] graphics package into a graph similar to Figure 4.1(c).

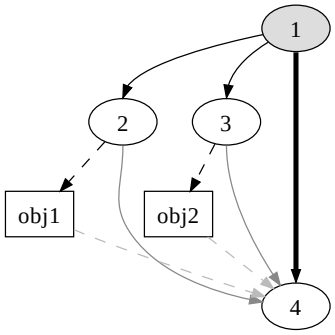
A ThreadScope graph G is a pair (V, E) where V is a set of vertices and E is a set of directed edges (u, v) between the vertices u and v where $E \subseteq \{(u, v) | u, v \in V \wedge u \neq v\}$. There are two types of vertices and three categories of edges. The vertices represent either serial execution blocks or memory objects. Execution blocks are graphically represented by round vertices, and memory objects are represented by rectangular vertices, scaled to represent their size. Each execution block is given a unique identifying number, as are objects. When the graph is drawn, the first execution block is colored gray to identify it. The edge categories are thread operations (spawns, joins, and continuations), memory operations (reads, writes, and atomic read/writes), and memory object identity transitions. Read and write edges come in a further two varieties, to distinguish atomic or synchronous operations from potentially unsafe operations. A thread, as ThreadScope defines it, is a sequence of execution blocks that are connected by thread continuations. Thread continuations are implicitly inserted whenever an execution block executes a potentially blocking operation

```
1  #include <cilk-lib.cilkh>
2  #include <stdlib.h>
3  cilk int genrand() {
4      return random();
5  }
6  cilk int main() {
7      int obj1 = spawn genrand();
8      int obj2 = spawn genrand();
9      sync;
10     return obj1 + obj2;
11 }
```

(a) Trivial Cilk Program

```
MALLOC now=3 addr=0x100 size=40
MALLOC now=5 addr=0x200 size=40
INIT now=10 tid=1 frame=0x100 threadid=0x0.0
INIT now=11 tid=1 frame=0x200 threadid=0x0.0
SPAWNED now=12 tid=2 frame=0x100 entry=1
SPAWNED now=13 tid=3 frame=0x200 entry=1
MWRITE now=22 tid=2 addr=0x104 size=4
MWRITE now=23 tid=3 addr=0x204 size=4
ENDED now=24 tid=2 frame=0x100 entry=1 next=2
ENDED now=25 tid=3 frame=0x200 entry=1 next=2
SYNC now=30 tid=1 threadid=0x0.0
MREAD now=32 tid=1 addr=0x104 size=4
MREAD now=34 tid=1 addr=0x204 size=4
FREE now=36 tid=1 addr=0x100
FREE now=36 tid=1 addr=0x200
```

(b) ThreadScope Event Description Log of Program in Subfigure (a)



(c) Graph Generated from Event Log in Subfigure (b)

Figure 4.1. Basic ThreadScope Stage Example

that necessarily establishes a “happens-before” dependency [102] on everything that follows it.

On these pages, the graphs are monochromatic, which can make distinctions between edge-types difficult to see. In practice, edges are presented in color. Here, thread operations are represented by solid edges. Thread continuations are represented by thick black edges, spawns by thin black edges, and joins by thin gray edges. Memory operations are represented by dotted or

dashed edges, for safe and unsafe operations, respectively. Reads are gray and writes are black. Atomic read/write operations are dotted with circles at both ends. Memory object transitions are thick, dashed, light gray lines.

For example, in Figure 4.1(c), round nodes 1 and 4 correspond to the `main()` function from Figure 4.1(a); node 1 represents lines 6–8 and node 4 represents lines 9–11. The `sync` operation in line 9 divides the thread into two execution blocks because it is a potentially blocking operation. Nodes 2 and 3 are both instances of the `genrand()` function, spawned in lines 7 and 8, respectively. They each write to a memory object (`obj1` and `obj2`) and exit. The `spawn` operations in lines 7 and 8 are indicated by the thin black edges of the graph, and the `sync` operation is indicated by the thin grey edges.

The graph is a progression through the logic of the program where parallelism is the x axis and the y axis represents logical ordering. The number of potentially concurrent actions at any point in the threaded program is equal to the number of solid lines or nodes at the y coordinate corresponding with the logical progression through the program. The maximum number of nodes or solid edges crossed by a horizontal line at that y coordinate is the maximum theoretical parallelism at that point.

4.2.1 Tracing

Event descriptions can be generated from a variety of tracing tools, from instrumented threading libraries, to runtime function call interceptors [127], to full instruction logs. ThreadScope's current set of event-collection tools are based on one of three data collection methods: instrumenting the threading library, system-level runtime tracing, and full instruction traces. In partic-

ular, there are implementations for the qthread library, Dtrace [24], Apple's libamber [6], and Sandia's SST simulator [148]. Because ThreadScope presents trace-based parallel application structure, it works best when that structure does not depend on the input. The parallelism presented in ThreadScope graphs is entirely dependent upon the parallelism requested during the instrumented application run. In a parallel environment like Cilk or pthreads that supports only explicit parallelism, all programmatic parallelism — the parallelism expressed by the programmer — is expressed at runtime and can be captured and expressed in a ThreadScope graph by a runtime tracing tool. Implicitly parallel environments, such as OpenMP [137] or UPC [55], usually adapt the programmatic parallelism to the available parallelism in ways that are not detectable at runtime without language-level instrumentation. Graphing the programmatic parallelism of an implicitly parallel environment would require generating the event description log at the level where the programmatic parallelism is visible, such as in the compiler or the threading library.

Each tracing method has both benefits and drawbacks. For example, an instrumented threading library can provide event tracing with relatively low overhead and can faithfully record all thread and synchronization operations. However, an instrumented threading library usually traces the entire execution of the program, which may not be desired. Additional functions to control tracing behavior can be added to the threading interface and program, but this solution would require modifying and recompiling the program in order to analyze it.

System-level runtime tracing, such as with valgrind [127] or Dtrace [24], provides the ability to track function calls or even track specific instructions.

The overhead of this type of tracing depends on how intrusive it is. For example, a Dtrace script can detect basic thread operations with relatively low overhead, and can be limited to tracing only a portion of an application's runtime. However, the utility of Dtrace event logs is limited because Dtrace cannot detect individual memory accesses. Examples of graphs based on Dtrace output are Figures 4.2, 4.3, and 4.4.

Full instruction tracers, such as Apple's libamber [6] trace generator or Sandia's cycle-accurate Structural Simulation Toolkit [148], record every instruction and can track every memory operation. This thorough data collection has a relatively high cost. Cycle-accurate simulation has the highest overhead, but avoids perturbing instruction ordering and thus can observe application behavior without affecting it. The event description in Figure 4.1(b) was translated from the verbose output of the SST simulator.

Every tracing technique has overhead associated with it. In many cases, as illustrated in Table 4.1, a great deal of overhead. The numbers in that table compare the execution time of each program run uninstrumented to the execution time with instrumentation. In most cases there is some additional post-processing time necessary to generate the event log from the trace outputs. The overhead of the tracing technique is primarily of importance when considering how long it will take to debug the program, it does not affect correctness unless the application being traced has strict timing requirements. This is especially true for cycle-accurate simulation, because the application being simulated is not aware of real wall-clock time, and the overhead of recording each instruction has no impact on execution order.

TABLE 4.1

TRACING OVERHEADS

COMPARED TO UNINSTRUMENTED EXECUTION

Benchmark	**Instrumented Thread Library**	**System-Level Tracing**	**Instruction Tracing**
Mantevo HPCCG	1.01x	22.39x	19698.79x
MTGL pagerank	3.49x	58.19x	8692.25x
piping	1.09x	72.85x	108.76x

4.2.2 The Event Description

Each event in ThreadScope's event language consists of a type and several attributes in key=value form. The basic thread lifetime events are `INIT`, `SPAWNED`, and `ENDED`, corresponding to when threads are allocated, run, and complete. Synchronization events include `LOCK`, `UNLOCK`, `SYNC`, `WAIT`, `INCR` and several others representing full-empty bit operations. Memory accesses are described by `MWRITE` and `MREAD` events. Unknown event types are ignored by the graph generator, thus allowing the event language to be expanded to support new types of analysis. For example, malloc-related events (`MALLOC`, `FREE`, and `REALLOC`) were added late in the development process to enhance memory object tracking. The event descriptions do not generally include data from within the threading libraries or system libraries. The event logs omit this information purposefully, to focus on thread-level application behavior.

Every event has a monotonically increasing timestamp, `now`, and a location or worker-thread identifier, `tid`. Other attributes depend on the event. Threads are uniquely identified by a tuple of their `frame` identifier — typi-

cally the address of the thread's bookkeeping structure or stack, which can be reused—and the timestamp that they began executing. For example, the `INIT` event indicates that a thread has been allocated. It has a `frame` attribute that indicates the identity of the thread being initialized. The `INIT` event specifies the identity of the thread generating the event with the `threadid` attribute; the default value for threads that are not spawned is `0x0.0`. The `SPAWNED` event indicates that a previously allocated thread has begun executing. This event defines a thread's identity (for future use in a `threadid` attribute), and so has three required fields: `now`, `tid`, and the relatively unique `frame` attribute. It has one optional field, `entry`, used for threading environments that allow for continuations. The `ENDED` event indicates that a thread has stopped executing. It requires the `now` and `tid` fields, as well as a `frame` field and an indication whether the thread is expected to continue. This indication is an optional `next` field that specifies what `entry` number the frame will next use. Subsequent `SPAWNED` events are considered to be continuations of previous threads if their `frame` and `entry` values match the `frame` and `next` values of an `ENDED` thread.

Memory is tracked by its address, and so synchronization events and memory accesses require an `addr` attribute. However, memory is typically treated as a collection of logical storage "objects" rather than as a large set of sequentially addressed one-byte storage units. Thus, ThreadScope tracks the threads and memory objects used in an application as objects with relationships to each other to simplify application structure. Memory addresses can be grouped into objects, for example as the result of `MALLOC` events.

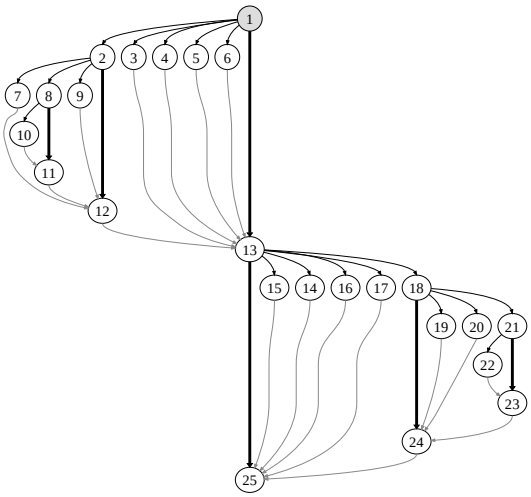


Figure 4.2. Structure of a Cilk Application With a Bottleneck
(without memory references)

4.2.3 Visual Representation

The first component of visualizing the structure of an application is to represent the relationships between its threads. Very simple relationships were illustrated in Figure 4.1(c).

A graph of this nature, for a given execution of a parallel application, can be generated for any threading mechanism that has uniquely identifiable threads. It demonstrates the logical connections between threads, and can be useful for identifying potential bottlenecks in the application. Regions where there is no available parallelism are bottlenecks that can be visually identified using this graphing technique. For example, Figure 4.2 depicts a simple threaded application with an obvious bottleneck.

The graph in Figure 4.2 does not indicate the severity of the bottleneck; it may be the synchronization and respawning of more threads or something more computationally intensive. The graph only reveals that there is a section

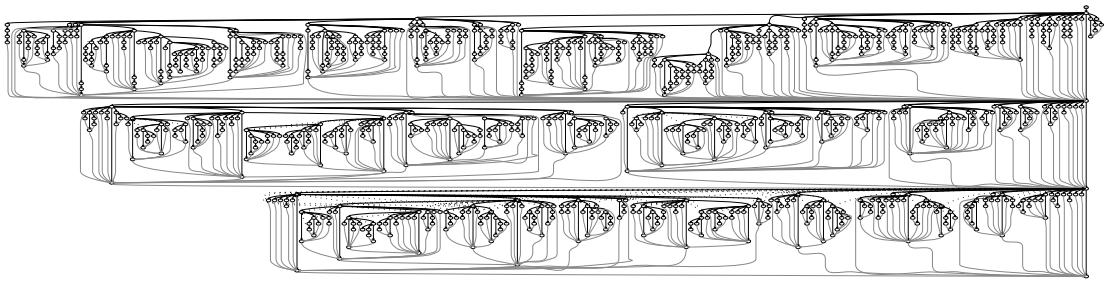


Figure 4.3. Structure of Cilk Bucketsort
(overview without details)

of the program (node 13, near the center) that cannot execute in parallel. That information is often a fact worth investigating when attempting to improve application performance. Figure 4.2 is a small excerpt from a Cilk application performing a parallel bucket sort. The full graph of this application is presented in Figure 4.3. Even without reading the source of the application, it is clear that it has two bottlenecks of the sort illustrated in Figure 4.2. These bottlenecks segment the computation into three parallelized segments and four purely serial segments.

4.3 Memory Access Patterns

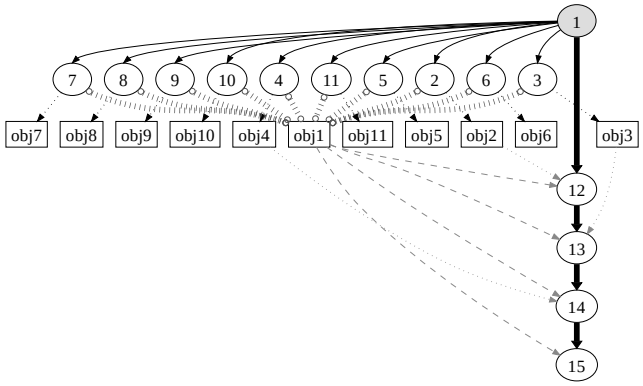
Once the structure of an application has been analyzed, the next step in performance and correctness analysis is to examine the program's memory access patterns. Even small programs generate a large volume of memory references. Including them all in a graph would make it dense and difficult to analyze. Making sense of the graph requires displaying only the memory references that are meaningful, and including them in the graph in ways that are helpful to the problem that is being analyzed. Exactly which memory refer-

ences are meaningful depends on the application and the situation, but there are some general simplifications that are frequently helpful in presenting a clearer picture of application behavior.

4.3.1 Improving Visual Clarity

One way to present a clearer picture is to eliminate from the graph all memory locations that are not written. This is useful for clearly presenting race conditions, programming errors, and synchronization bottlenecks. For example, Figure 4.4 illustrates the `qt_loop_balance()` function from the qthread library. The `qt_loop_balance()` function spawns a number of threads. Each thread, just before exiting, increments a shared counter with the atomic `qthread_incr()` function. The parent thread will wait for each thread to finish in turn. Each time a thread finishes, the shared counter's value is checked against the number of threads the parent spawned. If the counter's value is equal to the number of threads originally spawned, the parent can avoid checking the return values of the remaining threads.

In Figure 4.4, the dotted edges beginning and ending in grey circles are the atomic increment operations (`qthread_incr()`), the dotted black edges are synchronized memory writes (in this case, writing the return code of the function), and the grey edges (both dotted and dashed) are the relevant memory reads. In this case, a memory read is considered relevant if the read was either a blocking operation (`qthread_readFF()`; the dotted grey edges) or operated on an object that had previously been written to. The graph illustrates a `qt_loop_balance()` loop that spawned ten threads. Each thread wrote to both the shared counter and the thread's return-value location. The parent thread




```
long qt_loop_wrapper(qthread_t *me,
    const struct wrapper_args *arg)
{
    arg->func(me, arg->startat,
              arg->stopat, arg->arg);
    qthread_incr(arg->donecount, 1);
    return 0;
}
```

```
size_t iterend = start;
size_t each = len / NUM_SHEPHERDS;
int donecount = 0;
for (i=0;i<NUM_SHEPHERDS;i++) {
    array[i].startat = iterend;
    array[i].stopat = iterend += each;
    array[i].donecount = &donecount;
    array[i].arg = arg;
    if (extra > 0)
        { array[i].stopat++; extra--; }
    qthread_fork_to(qloop_wrapper, &(qwa[i]),
                    &(rets[i]), i);
}
for (i=0;donecount<NUM_SHEPHERDS;i++)
    qthread_readFF(me, NULL, &(rets[i]));
```

Figure 4.4. Structure of `qt_loop_balance()` Spawning Ten Threads with C Source Code

waited on three of the threads via a blocking operation (`qthread_readFF()`), each time checking the shared counter before waiting on the next thread. It waited on only three threads to finish (2, 3, and 4) before observing that the shared counter was the correct value.

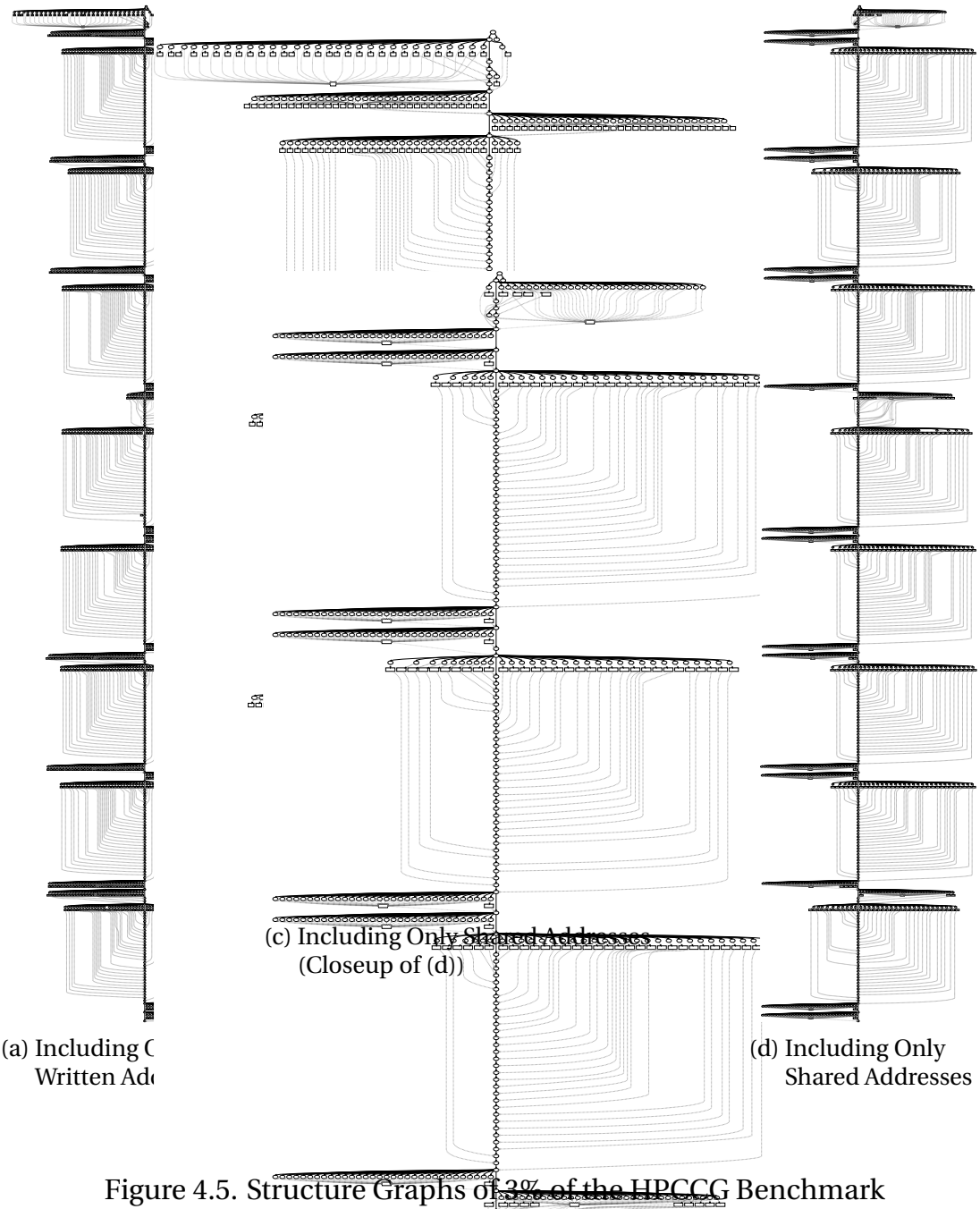
Figure 4.4 is a small snippet of a graph generated by the HPCCG [76] benchmark. A larger snippet, representing about 3% of its total runtime, is presented in Figure 4.5(a). This benchmark relies heavily on `qt_loop_balance()`. Since this function is used sequentially, and frequently, its serial components have the potential to become a bottleneck. Note that the structure of the program

can be observed in the memory and thread behavior, without requiring a priori knowledge of memory layout, data structures or the programmer's intent.

Considering only memory locations that are accessed by multiple threads is another useful simplification. For example, in a simple threaded matrix multiplication implementation, each memory location should only be written to by a single thread. If multiple threads write to the same memory location, that is probably a bug. Threads also typically use thread-specific scratch memory, often in the stack, that has no direct bearing on the logical structure or correctness of the application. The previous `qt_loop_balance()` structure simplifies slightly with this technique and makes the flow of information clearer, as illustrated in Figure 4.6. The HPCCG benchmark's graph is simplified this way in Figure 4.5(d).

4.3.2 Object Condensing

Programmers typically treat memory as a collection of logical objects rather than as a collection of sequentially addressed bytes. As such, displaying the logical objects rather than the addressed bytes simplifies the visual representation of memory operations. However, objects are hard to define conceptually; identifying them from an otherwise unidentified stream of memory addresses is virtually impossible. For example, an array may intuitively be an object, but it may be more useful in some situations to treat each element of the array as a separate object. More complex data structures only add to the problem. With limited a priori knowledge, the best approach is a winnowing process, consisting of successive identifications of important memory references from the set of unidentified references.



(a) Including C
Written Ad

(c) Including Only Shared Addresses
(Closeup of (d))

(d) Including Only
Shared Addresses

Figure 4.5. Structure Graphs of 3% of the HPCCG Benchmark
(overview without details)

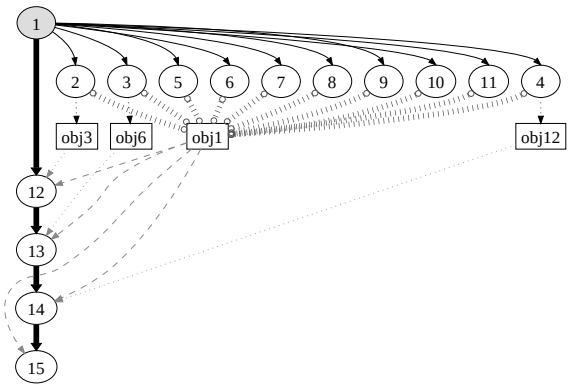


Figure 4.6. `qt_loop_balance()` Spawning Ten Threads, Memory Limited to Multiple-Accesses

In many cases, shared data structures are protected by synchronization operations of some kind, such as mutual exclusion locks, semaphores, or full/empty bits. In a very real sense, shared objects are defined by the locks that protect them, and obtaining the lock indicates that the lock's associated memory object is about to be accessed. Because of this behavior pattern, objects that are associated with specific locks can be extracted from a stream of memory address references by tracking the state of synchronization operations during memory access. In the case of mutex locks, whenever a thread accesses a memory location, that location is associated with whatever locks are currently held by the accessing thread. If an address is ever accessed without those locks, then it is not protected by them and cannot be considered part of the memory object associated with that lock. By the end of a sequence of memory references and mutex operations, each lock is definitively associated with the set of memory addresses that it protected during that sequence. That set of memory addresses comprises a single logical memory object. This object may not be contiguous and may not be entirely what the programmer ex-

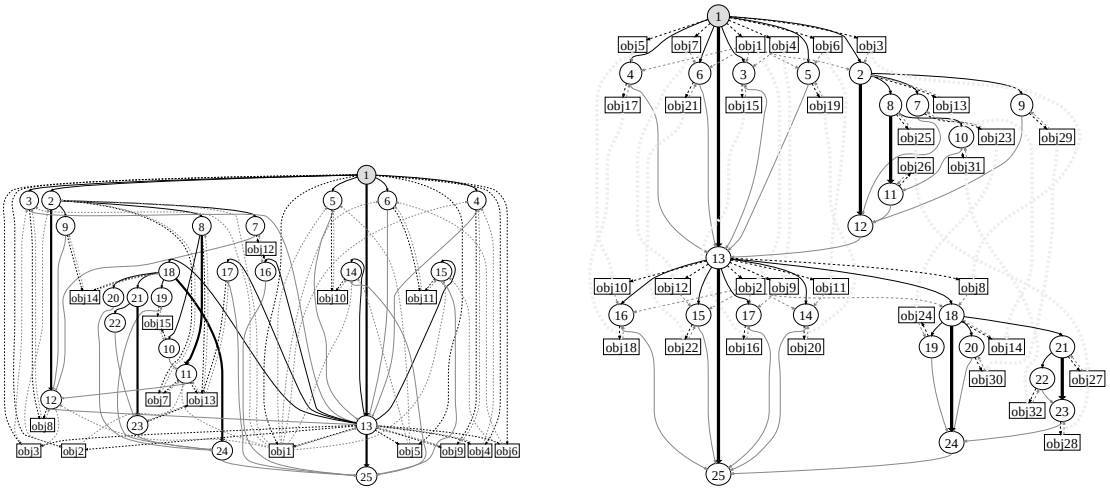
pected or intended, but it is a de facto memory object. Addresses that are accessed by multiple threads without a lock may indicate programming errors, and deserve closer attention.

Memory addresses that are not protected by synchronization operations can be clustered into objects in other ways. References to stack variables and global data can be identified not only by considering the memory region, but also with the aid of debugging information stored in the application binary.

Memory references into the "heap" region of the address space can be associated with the allocated region to which they belong — as defined by `malloc()`, `brk()`, `mmap()`, etc. — however that association is often insufficiently specific. Memory pools, arena allocation, and structured `mmap()`ed files are all situations where usefully identifying discrete memory objects can be especially difficult if they are not associated with synchronization operations. It is possible to take a probabilistic approach, estimating discrete memory objects with the help of proximity and temporal access patterns. However, without a priori knowledge of the application, grouping memory references into objects is, at best, a probabilistic guessing game. The only option that guarantees relative correctness is to assume that each unprotected memory location is an independent memory object — though the overabundance of tiny objects makes useful visualization challenging.

4.3.3 Memory Re-Use

Memory re-use impacts the observed structure of the application. Allocated memory and stack addresses are often reused, even though the object they represent has changed. It is generally considered good practice to reuse



(a) Naïve (b) SSA-Like Identity Tracking

Figure 4.7. Structural Impact of Memory Access and Identity Tracking

memory as much as possible to take advantage of processor caches. Thus, optimized algorithms typically reuse memory for unrelated computations. If the logical status of a memory object is not considered, structure can be difficult to extract; the threads will all appear to be operating on the same memory. For example, Figure 4.7(a) is the same program that was illustrated in Figure 4.2, but with memory references added. The logical structure so easily seen in the original graph has become hard to discern.

The most direct way of determining the logical identity of a memory object is to keep track of when it is allocated and deallocated: when it is deallocated, the memory is logically reset and if that memory is re-allocated, it clearly represents an entirely new logical object. Unfortunately, allocation and deallocation typically only apply to heap-type memory regions, and even then are not always easy to recognize—for example, in applications that implement mem-

ory pools or that simply re-use variables. Allocation and deallocation tracking, without the source-level modification to provide more information, is an incomplete and thus unreliable approach.

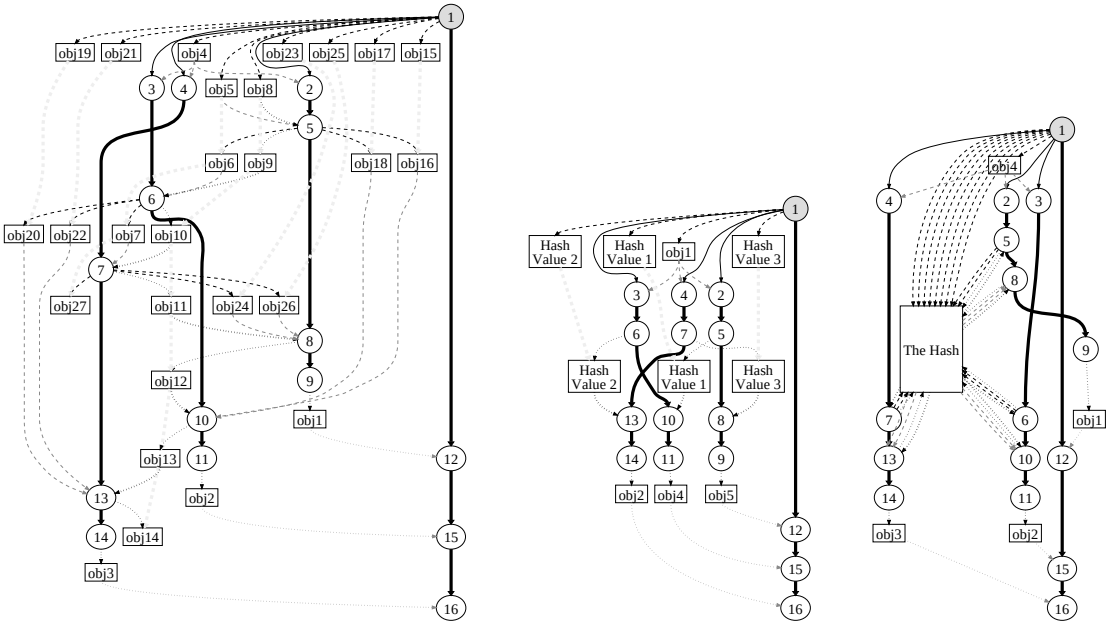
ThreadScope can also track the logical identity of memory by assuming that a memory block's identity changes when it is written. This approach is commonly known in the compiler community as Static Single Assignment (SSA) [41]. This divides a memory block's existence into separate identities much in the same way that threads are separated into connected execution blocks. The use of strict SSA to establish memory identity transitions means that each memory block may eventually obtain a large number of identities. Most of these identities are irrelevant to the overall flow of the application and can be merged together. Important memory references can be focused on by applying the previously discussed simplification heuristics, such as eliminating memory object identities that are not accessed by more than one thread. The operations upon and previous identities of a memory object impact whether an object must be considered to be shared by multiple threads. For instance, if a memory object is re-used by non-concurrent threads, the two instances are distinct only if the second thread writes to the object before reading from it. If the first action on the object is a read, the object must be treated as shared with the threads that had previously used it. Figures 4.7(a) and 4.7(b) represent the same program, but Figure 4.7(b) has a clearer structure because of this type of identity-tracking.

4.3.4 Condensing Structure with A Priori Knowledge

Not all memory references are equally important to analysis and debugging. For example, a shared data structure—such as a hash table or a linked list—may occupy a large discontiguous portion of memory. If that data structure and its accessor functions are assumed to be correct, or at least outside the scope of analysis, it can be beneficial to represent that data structure in the graph as a single object, rather than as a large set of independent memory locations.

ThreadScope's memory tracking can be modified by adding new events to the event description, thereby providing a priori information about the application's behavior. For example, `malloc()`-tracking, uses `MALLOC`, `FREE`, and `REALLOC` events to define memory objects.

Figure 4.8 illustrates the potential risks and benefits of redefining memory objects. Figure 4.8(a) is an example of a program with three threads that each insert an entry into a shared hash table and then get it back out again. In this case, the hash table is a simplistic one that allocates a separate object for each key/value pair with `malloc()`. Figure 4.8(b) presents the result of using memory allocaiton to define memory objects. The structure is relatively clear. Another way of condensing is to isolate objects by their operand functions. Figure 4.8(c) reduces the hash table to a single logical object that is accessed by multiple threads. Note that, like Figure 4.7(a), the logical structure of the graph is obscured by reducing the memory objects too far. Unfortunately, determining the existence of a hash table and isolating key and value pairs is difficult to do automatically from an otherwise nondescript address stream.



(a) Individual Memory References

(b) Condensing Malloc-defined Blocks

(c) Condensing Class-defined Blocks

Figure 4.8. Simple Hash Table Application, with Memory Object Condensing Options

Events representing hash table operations need to be added to the event log to mask the hash implementation's specific behavior.

4.4 Isolating Potential Problems

Identifying problems in parallel applications when there are few parallel threads of execution is not particularly difficult: the graphs have few components and the patterns of possible errors are easy to recognize visually. However, as the number of threads and the scale of the application increases so does the complexity and size of the thread structure graphs. Merely rendering the entire graph can be a challenge when analyzing an application that uses

thousands of threads. One powerful option for handling large graphs is the ZGRViewer tool [143]. ZGRViewer provides a quick way to navigate, magnify, and locate nodes in extremely large GraphViz-based graphs. Without such a tool, useful analysis requires that the volume of data presented in a single graph be limited to areas of interest, such as problem areas. There are two primary components to isolating potential problems in a thread structure graph: identification of areas of interest in the graph and selective display of only the portions of the graph relevant to that interest.

4.4.1 Structural Threading Problems

Some of the most basic problems that afflict threaded programs are structural problems that can be revealed and identified graphically. Problems such as race conditions and deadlocks are common errors that can often be discovered by analyzing the structure of the graph of the program.

4.4.1.1 Deadlocks

Tracking a deadlock down using a basic debugger can be an especially difficult exercise when there are a large number of locks involved. Deadlock is defined by the four Coffman conditions [31]:

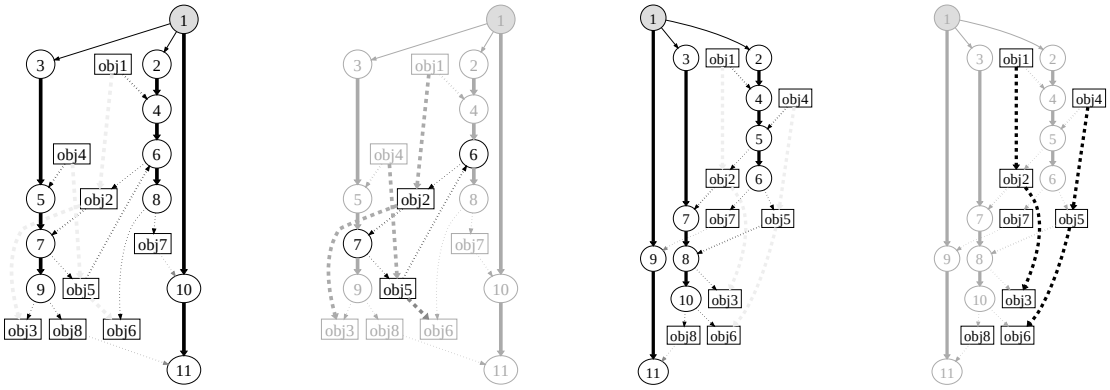
1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

In most multithreading programming models, the first three conditions for deadlock are assumed. The fourth, circular wait, is a structural description that becomes apparent from the thread structure graph of a deadlocked program, even if the program does not deadlock during execution.

Figure 4.9(a) presents a program that does not necessarily deadlock, but has the potential. In this program, two threads lock two locks. One thread (starting with node 2) locks the first lock (obj1/2/3), unlocks it, then locks and unlocks the second lock (obj4/5/6). The other thread (starting with node 3) locks the second lock (obj4), locks the first lock (obj2), then unlocks them in the same order. Because of the inconsistent ordering, this is a potential deadlock that may not occur at runtime. This can be detected with dependency tracking [157]. The structure graph can be interpreted as a resource-allocation graph that will have a circuit if deadlock can occur. Figure 4.9(b) highlights the circular dependency. Note that this program can (and did, during graph generation) run to completion, despite the potential deadlock, depending on how the threads are scheduled. Potential deadlock, however, can be identified with a depth-first traversal of the graph. When the previous program is rewritten to ensure that the locks are only obtained in a specific order, as illustrated in Figure 4.9(c), the circular wait is eliminated. The memory state transitions are highlighted to illustrate the lack of a cycle in Figure 4.9(d).

4.4.1.2 Race Conditions

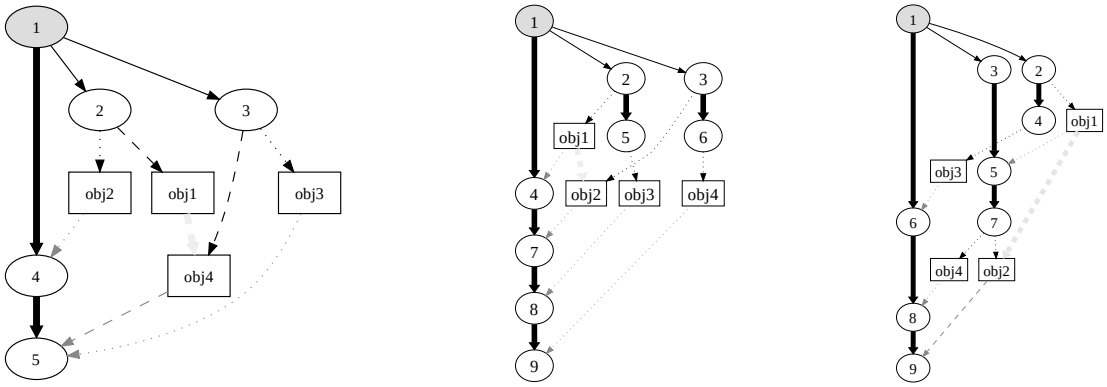
There are many different kinds of race conditions, but not all of them can be easily recognized by even the most advanced automatic analysis system. Basic race conditions, such as multiple threads manipulating the same mem-



(a) Circular Wait (b) Highlighted Circuit (c) No Circular Wait (d) Highlighted Dependencies

Figure 4.9. Identification of Potential Deadlock via Structure

ory object without synchronization, can be relatively easily identified. When the race is to see which piece of data will be placed in a thread-safe memory object, identifying the race condition can become more challenging. However, a graph of all of the relevant accesses to a given memory object can reveal the potential for race conditions. If multiple threads access the same memory object there is a potential race condition and source of concern. Figure 4.10 illustrates three different kinds of shared memory access. In Figure 4.10(a), two threads (2 and 3) attempt to write into the same memory object that thread 5 later reads. However, there is no required ordering to these writes, and thread 5's read could return either value written, or even some corrupt combination of the two. Figure 4.10(b) illustrates the common situation of a shared memory object protected by mutexes. This protection eliminates the potential for a corrupt data read, but does not establish a required ordering for the writes. The writes do not depend on one another, so they can be executed in any or-



(a) Unprotected Race (b) Protected Race (c) Ordered Access

Figure 4.10. Identifying Race Conditions via Structure

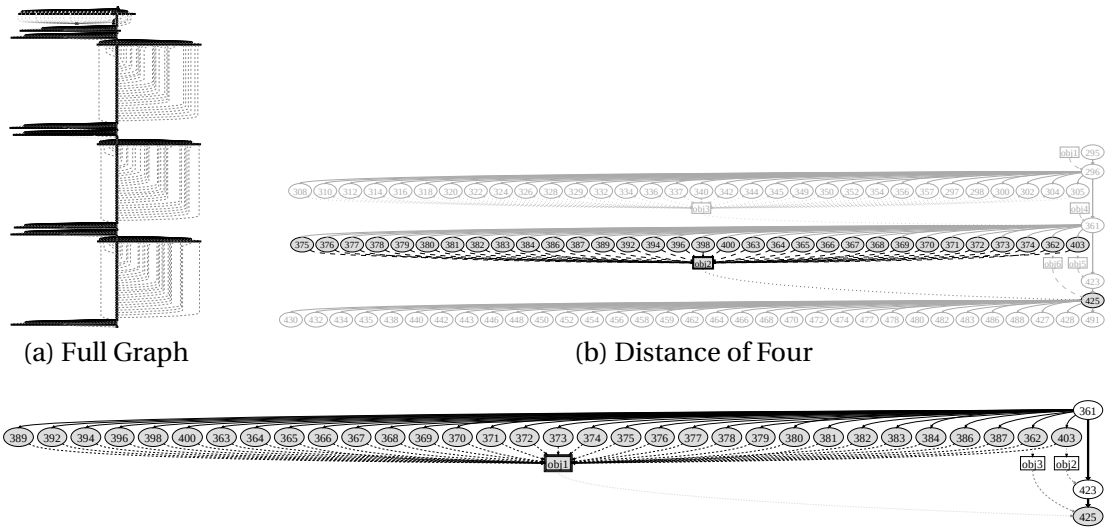
der. This nondeterminism may not be an error, depending on the application. Finally, Figure 4.10(c) illustrates shared access to a protected memory object that does not have a race condition. Because the writes are logically ordered through dependence relations, the contents of the memory object are deterministic. Thus, the final read is both safe and deterministic despite being unsynchronized. Because these graphs represent the logical structure of the program, a dangerous potential race condition (4.10(a)) can be identified even if no error is apparent at run time.

It is worth emphasizing that not all race conditions, defined as timing-dependent logic, are errors. For example, the correctness of the `qt_loop_balance()` implementation from Figure 4.4 does not rely upon a specific ordering of thread completions.

4.4.2 Graph-based Problem Isolation

The useful portions of the graph can be isolated by presenting only a subgraph of the total program structure containing just the areas of interest. For example, the graph can be reduced to the subgraph of only the nodes that are connected to a given thread or memory object. Because the graphs are directed graphs, it is possible to find the nearest common ancestor of two or more nodes and present only the subgraph of the paths from those nodes to their common ancestor. Figure 4.11 illustrates such graphs. Figure 4.11(a) is a graph of a short application with an intentional race condition in it. Figure 4.11(b) narrows the graph of the application to only the nodes that have a distance of four or less from the memory object with the race condition—the nodes that are directly connected to the memory object are highlighted. Figure 4.11(c) presents the graph of the threads that touch the memory object of interest and the ancestral tree up through the nearest common ancestor of those threads. Both of these presentation modes are useful for visually locating potential structural problems.

The other aspect of debugging is identifying the problems in a large graph algorithmically so that they can be isolated and displayed. This is where heuristics are useful, similar to standard compiler warnings. One common structurally-detectable race condition is where a write occurs to an object that has not necessarily been read yet. A race condition also occurs when there are two writes to a memory location that do not depend on each other, which can be identified algorithmically. When a deadlock occurs, of course, the affected threads and memory operations can be identified, isolated, and displayed.



(a) Full Graph

(b) Distance of Four

(c) Nearest Common Ancestor

Figure 4.11. Race Condition Isolation: Presentation Options

Identifying potential deadlocks is also possible to do algorithmically, as has been discussed.

4.5 Parallel Computation/Communication Models

One of the particularly interesting aspects of this kind of multiprocessing analysis is that the programming scheme employed by the parallel algorithm being studied can be observed and understood without in-depth knowledge of the program itself. The computation model and communication patterns used by the application impact the performance characteristics of the application, and provide an indication of likely performance trends. The computation model is closely associated with the communication pattern and provides insight into potential optimizations and problems that can assist in debugging and maintenance.

For example, graphs in Figure 4.5 were generated from the HPCCG benchmark, which was modified to use qthreads in Chapter 3. Because of the naïve parallelization approach used, HPCCG exhibits a distinctly phase-oriented programming model that is comparable to the Bulk Synchronous Parallel [160] and PRAM [60] computation models. In each parallelized segment of the application, threads are created, execute, and then communicate and collect results, largely in the form of synchronization operations. The structure of these computational segments can be viewed more closely in Figure 4.4, which illustrates a single instance of the qthread-based parallel loop construction used in HPCCG.

The bucketsort implementation, graphed in Figure 4.3, is an example of a distinctly different parallel computation model. While the program is obviously composed of three distinct phases, without memory references, the memory model cannot be determined. The graph in Figure 4.3 was produced with a Dtrace-generated event description, which could not detect memory references. Figure 4.12 is a graph of 10% of the same bucket sort program, but traced with SST in order to include memory references. Predictably, it is centered around the large array that it is sorting, depicted as the large box near the top of the graph. This behavior makes it similar in some ways to a Linda-based application [4]. The same would be true of most parallel applications centered around a single data structure, though some data structures can be graphed more usefully, such as a hash table, as illustrated in Figure 4.8 and discussed in Section 4.3.4.

Flow-based applications [121] have another distinct structure. This structure is illustrated in Figure 4.13, which is a graph of a simple parallel stream

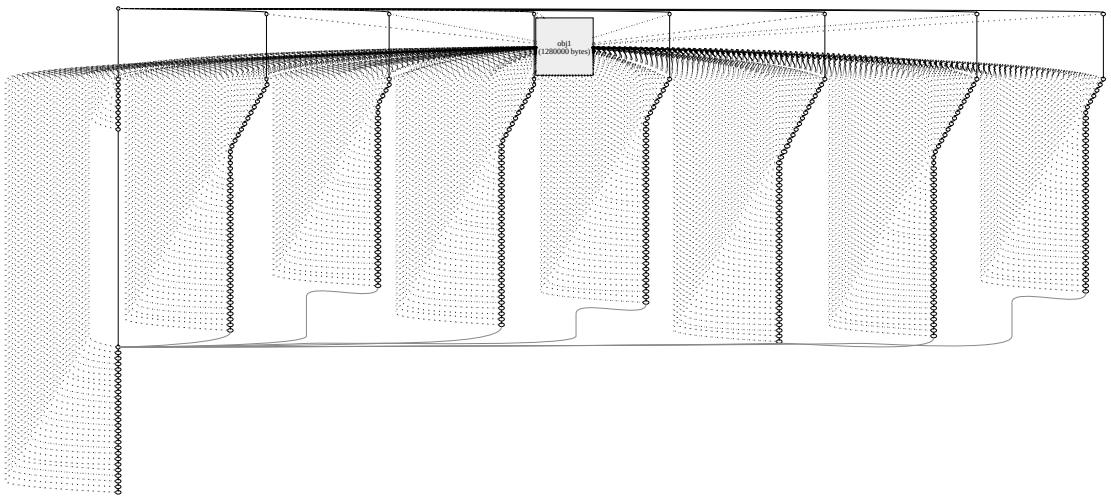


Figure 4.12. Structure of 10% of Cilk Bucketsort, Including Memory References

processor. This program spawns four threads. The first thread generates random numbers and puts them into a circular buffer. The second thread fetches numbers from that circular buffer and feeds only the odd numbers into a second circular buffer. The third thread fetches numbers from the second circular buffer, sorts them, and then puts them into a third circular buffer. The fourth thread fetches numbers out of the third circular buffer and prints the unique ones. All three circular buffers have a capacity of three. Note that, rather than rely on a large set of shared objects that multiple threads can access, each shared memory object is only accessed by two threads. This leads to a distinctive visual pattern. The resulting thread structure graph has some distinct similarities to the corresponding FBP diagram.

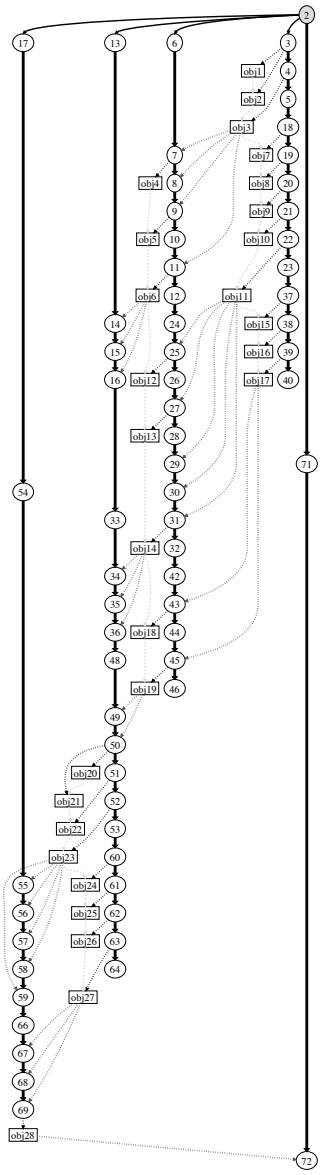


Figure 4.13. Structure of a Flow-based Application

4.6 Conclusion

Analyzing parallel applications continues to be an area of great interest as parallel runtime environments become more powerful, complex, and unpredictable. The work presented in this chapter provides a powerful tool to assist in understanding the behavior of large-scale threaded applications in lightweight threading environments. This allows application structure to be compared across multiple threading environments and assists in quickly identifying hard-to-reproduce logical problems. Most importantly, this work allows the memory use patterns and thread structure to be combined in a single visualization tool, enabling not only correctness analysis but providing the information necessary to plan thread/data partitioning schemes.

CHAPTER 5

EXPLOITING MACHINE TOPOLOGY WITH ADAPTIVE DISTRIBUTED DATASTRUCTURES

5.1 Introduction

The previous chapter discussed application structure, both how it can be used to identify parallel programming errors and how it can be used to understand an application's behavior. Because shared memory parallel machines have non-uniform memory access latencies with a variety of topologies, application structure and how it maps to memory topology directly impacts parallel performance. This chapter explores this relationship within the context of commonly used data structures. Data structures provide a clear illustration of the connection between structure and performance in parallel systems. In particular, data structures' internal structure is defined by data placement.

Three new data structures — a distributed memory pool, a distributed array, and a distributed queue — that adapt to memory topologies are presented along with benchmark results demonstrating their efficiency. The distributed memory pool proves to be up to 155 times faster than the standard `malloc()` implementation while providing location-aware memory allocation. The distributed array supports strong scaling, providing a 31.2 times performance improvement with 32 ccNUMA nodes while iterating over the array. At scale, the

distributed queue demonstrates up to a 47 times improvement over a strictly-ordered lock-free queue, and an 8.3 times improvement over the performance of state-of-the-art concurrent queues on a large ccNUMA system.

The benchmark results presented in this chapter to demonstrate the scalability of these new data structures are gathered from several different parallel architectures with a variety of memory topologies: a four-core Xeon workstation, a 32-core Niagara 2 server, and a 48-node SGI Altix. By comparing operations-per-second and effective bandwidth, the benchmarks also demonstrate the importance of choosing appropriate system-specific design parameters, such as memory distribution pattern and segment size. Distributed data structures that take advantage of locality and adapt to system topology solve several of the basic problems of increased parallelism in complex systems.

The primary challenge of implementing portable distributed data structures is supporting multiple systems and topologies. The qthread library supports many different operating systems and hardware architectures and provides a consistent interface to their unique methods of discovering machine topology and specifying the location of memory and computation. Fully exploiting vastly different parallel architectures requires adapting to the available hardware features and choosing data distribution patterns and location assignment mechanisms that suit the system topology. The distributed data structures presented use the locality information provided by the qthread library to select distribution strategies and to optimize communication patterns. They also rely on the qthread library to provide access to fast atomic synchronization operations.

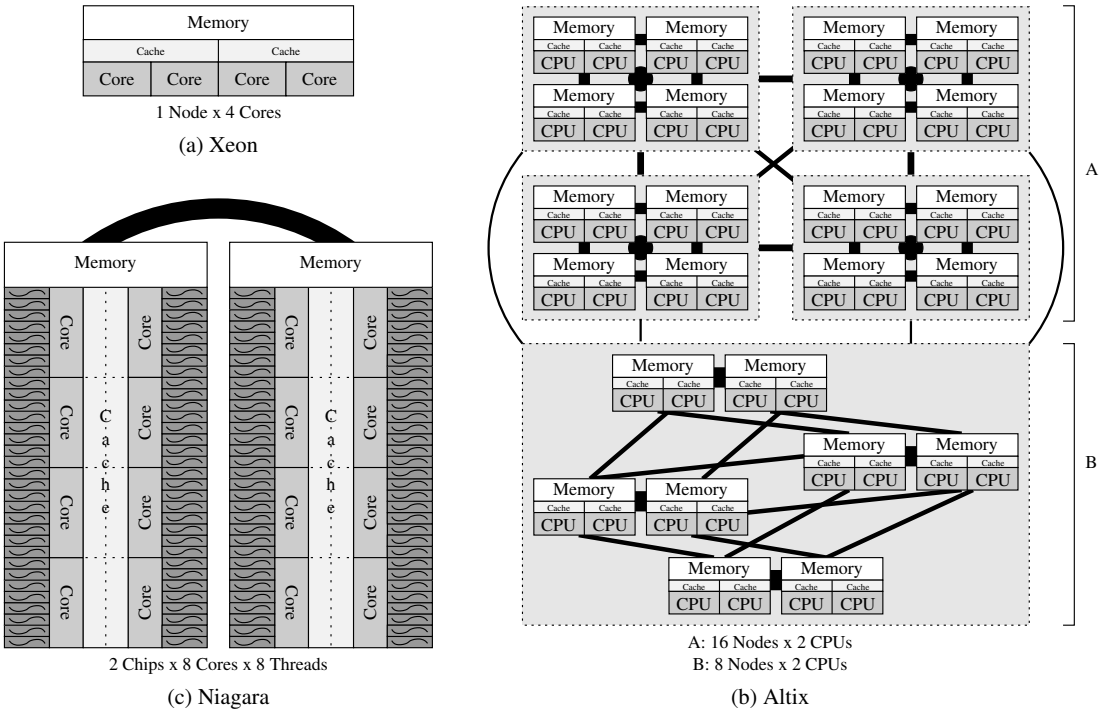


Figure 5.1: System Topologies

5.2 Parallel Architectures

Several different systems with dramatically different topologies are used to evaluate the distributed data structures in order to demonstrate using topology to inform runtime decisions. These systems include a dual-processor dual-core Intel Xeon 5150 workstation, a 48-processor SGI Altix 3700 ccNUMA SMP, and a dual-processor 16-core Sun SPARC Enterprise T5240 server. These machines were selected to demonstrate three different types of parallel systems: a common development workstation, a large ccNUMA system with a wide range of memory latencies and a complex topology, and a massively multithreaded high-throughput chip architecture. The topology of each of these machines is illustrated in Figure 5.1.

The Intel Xeon workstation, Figure 5.1(a), is dual-processor and dual-core, typical of commodity development workstations. Each processor has its own cache but shares the 1066 MHz front-side bus to memory. The system runs Linux, which provides the libnuma [96] interface for querying system topology and assigning location to threads and memory blocks. Because the processors share the front-side bus to memory, the libnuma library describes this system as a single node with four processors. This lack of detail is unfortunate, since the cache-sharing is an important aspect of the memory hierarchy. However, ignoring cache coherency issues, the memory latency is roughly the same for both processors.

The Altix ccNUMA SMP, Figure 5.1(b), is based on the Intel Itanium 2 architecture. The nodes are connected via NUMAlink 4 with dual 3.2 GB/s unidirectional links. Each of its 24 processing nodes has two CPUs and a large amount of local RAM. The machine is divided into two components: a 16-node component (A) and an 8-node component (B). Component A consists of four clusters of four dual-processor nodes, for a total of 32 processors. Component B's nodes are arranged in a dual-plane fat-tree. The two components are connected by additional dual unidirectional links, though the node-pairs in component B are not both connected to the same set of nodes in component A and some of the links between the two components are asymmetric. This system also runs Linux, and provides the libnuma interface. The libnuma library describes this system as 24 nodes, each with two CPUs.

The Niagara 2 server, Figure 5.1(c), exemplifies a recent direction in multiprocessor systems targeted at the high-throughput server market. It has two processors, each with eight cores. Each core can support up to eight concur-

rent hardware threads, for a total of 128 concurrent hardware threads. Each core has its own bank in the L2 cache, but any bank in the cache of a single processor can be accessed by any other core on the same processor. Each pair of cache banks shares a dual channel FB-DIMM memory controller. This system runs Solaris 10, 5/08 Release, which provides the liblgrp [167] interface for querying system topology and assigning location to threads and memory blocks. The liblgrp interface presents a coarse hierarchical view of the system. The smallest granularity of topological detail provided by this library is a single "locality-group", which may contain multiple cores and multiple threads. Though there is no inherent limitation to the granularity of a "locality-group", in current Niagara 2 systems the smallest locality group refers to an entire processor.

In the absence of a direct representation of topology, such as liblgrp provides, topology can be inferred from measurements of distance or latency between components. However, distance or latency cannot be reliably determined by measuring the access time of addresses throughout memory space. Ignoring issues of the virtual memory system remapping pages, the memory access latency can vary every time it is measured due to the interference of prefetchers, cache coherency protocols, and other programs in the system, among other things. The topology libraries used by qthreads — Linux's libnuma and Solaris's liblgrp — provide access to stable, albeit unspecific, latency measurements. The libnuma measurement of "distance" between nodes is a unit-less number generated by the Linux kernel from system-specific sources. All inter-node distances are normalized to the speed of "local" memory, which is defined to be 10 units away. Liblgrp provides a latency measurement in

unspecified machine-specific units that is not guaranteed to represent actual latency. Conveniently, liblgrp also provides a hierarchical representation of the system to directly establish topology. The qthread library uses these interfaces to establish, for each location, an ordered proximity list of locations and access to a stable measurement of distance between locations. However, to avoid imprecision, the qthread library uses an untranslated measurement from whichever system-specific library is available. When a locality interface is not available, all locations are assumed to be equidistant.

5.3 Distributed Data Structures

Three new data structure designs demonstrate the benefits of adapting programmatic structure to machine topology: a pool, an array, and a queue. These data structure types are cornerstones of basic parallel application design, and are used as basic building blocks in a wide range of high-performance applications. The key design point of these data structures is the way that they adapt to the topology of the system in use. The API of the data structures is summarized in Figure 5.2.

5.3.1 Distributed Memory Pool

Memory allocation is a frequently overlooked detail of large programs that can significantly impact performance. Standard memory allocation libraries are typically designed for general-purpose allocation in single-threaded applications, balancing allocation speed with the need to limit fragmentation, without concern for locality. The libnuma `numa_alloc()` function provides a way to allocate memory in a specific location, but at a heavy cost: it is de-

Memory Pools	Arrays	Queues
qpool_create(*size***)** Create a pool of *size*-byte objects **qpool_create_aligned(***sz***,***align***)** Create a pool of *sz*-byte objects, aligned to *align* byte boundaries **qpool_alloc(***pool***)** Fetch an object from the *pool* **qpool_free(***pool***,** *addr***)** Return *addr* to the *pool* **qpool_destroy(***pool***)** Deallocate *pool* and all of its memory	**qarray_create(***cnt***,***size***)** Allocate an array of *cnt* elements, each at least *size* bytes, and distribute its memory **qarray_create_tight(***cnt***,***size***)** Allocate an array of *cnt* elements, each exactly *size* bytes, and distribute its memory **qarray_elem(***array***,***n***)** Locate the *n*'th element in *array* **qarray_elem_migrate(***array***,***n***)** Locate the *n*'th element in *array*, migrates thread near to that element **qarray_iter_loop(***array***,***f***,***arg***)** Execute the function *f* on each element in *array* in parallel **qarray_destroy(***array***)** Deallocate *array*	**{qlf|qd}queue_create()** Allocate a queue **{qlf|qd}queue_enqueue(***q***,** *elem***)** Append the element *elem* to the *q* queue **{qlf|qd}queue_dequeue(***queue***)** Get the head of the *queue* **{qlf|qd}queue_empty(***queue***)** Check whether *queue* contains any elements **{qlf|qd}queue_destroy(***queue***)** Deallocate *queue* *A qlfqueue is a lock-free queue, and a qdqueue is a distributed queue.*

Figure 5.2. Distributed Data Structure API (Abridged)

signed to allocate multiple pages of memory at a time, rather than as a general purpose allocator. Liblgrp allows any allocated memory page to be assigned a location via `memadvise()`, but this interface is also restricted to specifying the location of entire memory pages.

5.3.1.1 Design

A simple mutex-protected memory pool could be used with `numa_alloc()` or `memadvise()` to provide thread-safe access to small blocks of location-specific memory, however the overhead of mutexes can be significant. One way of minimizing mutex overhead is to use a lock-free algorithm — where all states of the object are consistent, state transitions are made via atomic hardware operations, and forward progress can always be made. The qthread memory pool, qpool, is primarily implemented as a set of location-specific lock-free

stacks, using qthread's `qthread_cas()` atomic compare-and-swap operation. Using separate pools for each location in the system provides fast access to location-specific memory, but can exacerbate problems of imbalanced memory allocation patterns. Memory may be allocated in one location and deallocated in another, causing memory to be "misplaced," and thereby wasted. Thus, it is important to be able to locate the "home pool" of each allocated block of memory so that it can be returned there when deallocated. Topology information can also be used to pull memory blocks preferentially from nearby memory pools rather than either pulling from distant pools or allocating new memory.

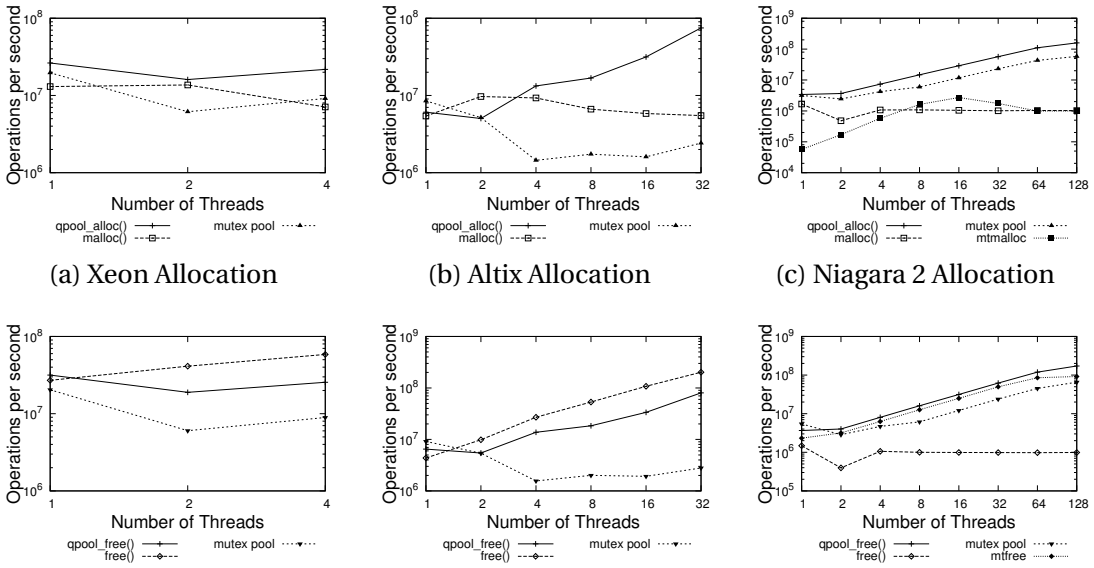
Before a qpool can be used, it must be created and initialized with a call to `qpool_create()`. Thereafter, elements can be fetched from the pool using the `qpool_alloc()` function and returned to the pool using the `qpool_free()` function. A pool may be destroyed using the `qpool_destroy()` function. Internally, each allocation operation is handled by the pool specific to the location of the requesting thread.

Memory exhaustion is of particular concern when allocating node-specific memory, because each node has only a portion of the overall memory. There are two ways of handling node-specific memory exhaustion when allocating memory: fail to allocate memory, or attempt to satisfy the request by violating the location requirement. The qpool takes the latter approach, and fails to allocate memory only if memory cannot be allocated on any node. Attempting to create pools of extremely large memory objects will also fail; when creating the pool, blocks of memory are allocated for each shepherd. If this memory cannot be allocated, the pool creation fails, and the `qpool_create()` function returns a null pointer.

When memory must be allocated for the pool's back end, it is allocated in large blocks and portioned out in chunks of the size specified when the pool was created. Memory that is freed is pushed onto a location-specific re-use stack. When memory is requested, this stack is checked first. If the re-use stack is empty, memory is pulled from the local large allocated block. When this block is exhausted, the re-use stacks of the closest neighboring locations are checked, in random order. If none of them have memory, a new large block of memory is allocated from the local node's memory, and added to a list of allocated blocks. If allocation fails, the re-use stacks of all pools are checked, in order of their distance from the requester. Finally, all pools are checked for large allocated blocks. If no memory can be found, the allocation request will return a null pointer.

5.3.1.2 Benchmark

To illustrate the scalability of the qpool's design, a benchmark was designed that uses multiple threads to allocate and write to 100 million separate 44-byte memory blocks (the size of a `pthread_mutex_t` on some systems) before deallocating them. Three allocation methods are compared on the Linux systems: the qpool lock-free memory pool, a similar concurrent pool using pthread mutexes instead of atomic operations, and standard glibc malloc [65]. Solaris also provides a multithreaded malloc library, mtmalloc, which is also tested. Each set of allocation and deallocation operations is performed ten times on each of the three test systems.



(a) Xeon Allocation (b) Altix Allocation (c) Niagara 2 Allocation

(d) Xeon Deallocation (e) Altix Deallocation (f) Niagara 2 Deallocation

Figure 5.3: Memory Allocation/Deallocation with 100 Million Elements

These graphs compare the allocations and deallocations per second of multiple threads allocating 44-byte blocks of memory. Each block is written to, ensuring that it has been allocated. Freeing memory with standard malloc is extremely fast on Linux systems, but qpools are up to 155 times faster at scale on the Niagara 2. Mutexes have a high cost on Linux.

5.3.1.3 Results

The graphs in Figure 5.3 compare the allocations and deallocations per second of the benchmark application. To make the different allocation and deallocation methods easier to differentiate, the y-axis of the graphs do not go to zero.

As the number of concurrent threads increases, particularly beyond four threads, the qpool outpaces glibc malloc. The mutex-based location-specific memory pools suffer from slow mutex operations on Linux. The glibc malloc implementation, while not returning location-specific memory, achieves a great deal of speed with its adaptive arena allocation. It avoids contention by creating additional allocation arenas.

The Solaris malloc library (used in Figures 5.3(c) and 5.3(f)), designed for serial applications, is uniformly slow. The Solaris multithreaded malloc implementation (mtmalloc) provides somewhat better performance for relatively low numbers of threads, but does not appear to be designed to support more than sixteen threads at a time. Unfortunately, mtmalloc has a high memory cost, allocating only blocks of memory in sizes that are powers of two.

5.3.2 Distributed Array

Distributed arrays are commonly used as a central data structure in scientific computing, particularly as matrices or to efficiently store large amounts of similar data. In some cases, such as Co-Array Fortran, the distributed array is the fundamental metaphor for and arbiter of parallel operation.

5.3.2.1 Design

The design of the qthread distributed array, or "qarray," is a basic "blocked" or "tiled" array design: the array is broken into segments that are placed in different locations around the system. This section examines three aspects of this distributed array design: distribution pattern, segment size, and element size.

The qarray distributes its memory when it is created, via the `qarray_create()` function. Once it has been created, elements within the array can be accessed with either an accessor function, `qarray_elem()`, or using pointer math within a segment. There is an alternate accessor function, `qarray_elem_migrate()`, that also migrates the caller to be near the specified array element. The qarray provides an efficient mechanism for iterating

over the array: the `qarray_iter_loop()` function. This mechanism calls a user-specified function to process ranges of the array, and ensures that the function will be executed close to the range that it processes. Qarrays are deallocated with the `qarray_destroy()` function.

The distribution pattern of array segments, and the method of determining storage location, necessarily impact performance. One simple distribution mechanism is to place each segment according to its order in the array, via a hash or other mathematical mapping. This has the virtues of simplicity and uniformity, and avoids accessing memory to locate segments. This method is, however, fixed and unadaptable, and would be unacceptable if relocating array segments was necessary. Alternately, the location of each segment can be stored with the segment. This requires accessing the segment to discover its location, but enables both a wider range of distribution patterns and relocating segments at runtime. Several options are considered:

Static Hash uses the segment order number, modulo the number of shepherds, to determine the location of the segment.

Dist Reg Stripes stores segment location with the segment, but distributes memory similarly to the Static Hash.

Dist Rand stores segment location with the segment and distributes segments randomly.

Dist Reg Fields stores segment location with the segment and clusters sequential segments onto the same shepherd, distributing segments evenly.

Serial Iteration is not a distribution pattern, but is used to show the relationship of the distributed iteration bandwidth to the bandwidth achieved

with a typical non-distributed array and iteration loop, optimized by the compiler.

5.3.2.2 Benchmarks

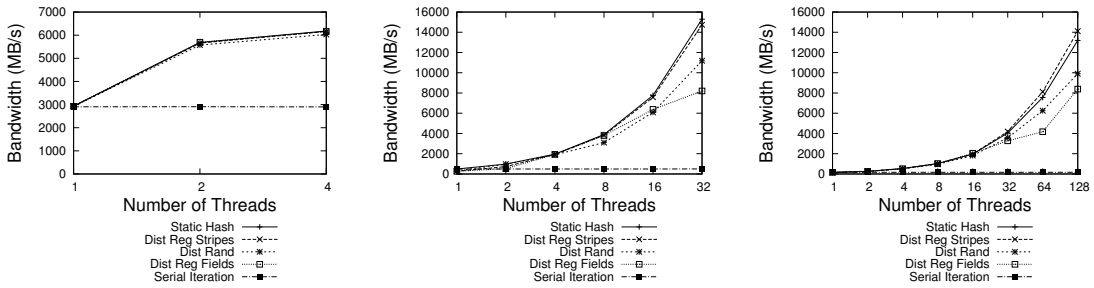
Two benchmarks were designed to explore the performance characteristics of the qarray distributed arrays. The first benchmark creates a 100-million element qarray of either 10,000-byte elements, 40-byte elements, or 8-byte elements, using each of the previously described distribution methods. The elements are all initialized. The benchmark then uses the qarray parallel iteration function to read the values of all of those array elements. The size of the array divided by the time required to test each value represents the effective memory bandwidth achieved. Several versions of this benchmark were compiled, each using a hardcoded number of pages for the internal array segment size.

The second benchmark similarly uses all of the available distribution methods in turn, but rather than using fixed sizes, uses arbitrary size elements that are either "packed" tightly, or are allowed to have their size rounded to the next-highest multiple of eight.

5.3.2.3 Results

Figure 5.4 presents the results of the first benchmark, using 8-byte elements, treated as double-precision floating point numbers. The benchmark was run ten times on each system. The results presented represent the average bandwidth achieved.

It is worth noting that the parallel iteration technique scales well in all three systems, but the maximum performance depends on the memory configura-



(a) Xeon (b) Altix (c) Niagara 2

Figure 5.4: Distribution Pattern Scaling

These graphs compare the memory bandwidth achieved by multiple threads iterating over arrays of 100 million double-precision floating point numbers, reading their values. Multiple distribution patterns are compared to serial execution. Maximum performance depends on memory configuration, but in general, some variety of striping provides the best performance.

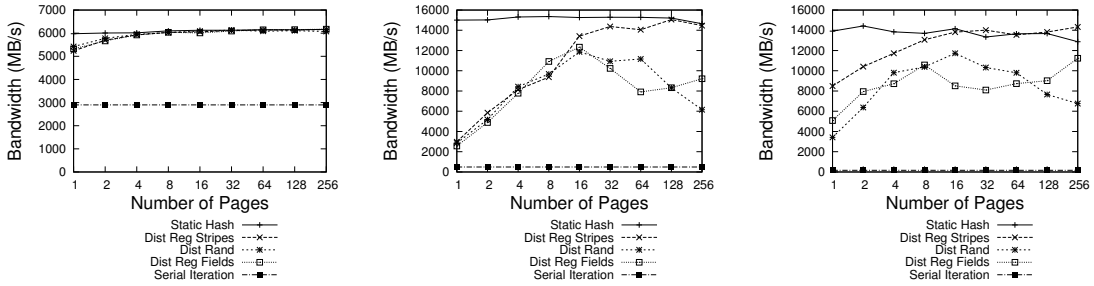
tion. On the Niagara 2 server, iteration with 128 threads operated 86.2 times faster than serial iteration. On the Altix machine, iteration with 32 nodes operated 31.2 times faster than serial iteration. The Xeon workstation peaked at 2.1 times faster than serial operation, likely because there are only two processors, and thus two memory controllers, which compete for a relatively low-bandwidth memory bus.

On both the Altix and the Xeon, the static hash was the fastest distribution pattern. This is almost certainly because accessing remote memory to determine its location is slow and can cause incorrect memory blocks to be prefetched. Because their cache hierarchy is characteristic of most commodity systems, the static hash is the default distribution method for all systems other than the Niagara 2. The Niagara 2 server benefits from fetching "incorrect" blocks because of its cache sharing. Though a fetched block may be the incorrect one for one thread, it is likely to be useful for another nearby thread. The Dist Reg Stripes distribution pattern leverages the Niagara 2's

shared cache by interleaving segments and thereby clustering the working set of memory, resulting in the best performance.

Distributed array performance also depends on the granularity of distribution, or "segment size." Most locality-aware memory allocation methods operate on memory blocks no smaller than a page, requiring that all memory within a page be in the same place, thereby defining the minimum segment size. While it is certainly possible to create a fine-grained distribution pattern by striping array elements across multiple pages such that each page contains non-consecutive elements, this prevents efficient loop unrolling when iterating over sets of data. Because the minimum segment size is a page on all of the supported systems, only large arrays can be efficiently distributed. The previous benchmark was recompiled with several different segment size options hardcoded into the array implementation.

Figure 5.5 illustrates the impact of segment size on bandwidth use and scaling. When iterating through a qarray in parallel, large segment sizes allow the machine to take advantage of cache features such as prefetching. At the same time, large segment sizes can also create load imbalances and limit efficient fine-grained parallel operation within those segments. The distribution method has a significant impact on the optimal segment size. The performance of static hash across the range of segment sizes varied less than 12% on the Niagara 2 server, while the best performing segment size for the Dist Reg Fields distribution provides a 220% improvement over the worst performing segment size. The static hash distribution provided the best small segment size performance of the distribution methods tested. Other distribution methods need larger segments to fully amortize the cost of incorrect prefetching. In



(a) 4 threads, Xeon (b) 32 threads, Altix (c) 128 threads, Niagara 2

Figure 5.5: Segment Size's Impact on Scaling

These graphs compare the bandwidth achieved by using multiple threads to iterate over arrays of 100 million double-precision floating point numbers, reading their values. Multiple distribution patterns and segment sizes are compared to naïve serial execution. Each architecture and distribution pattern has its own optimal segment size, but 16 pages is a good default.

most cases, a segment size of 16 pages is close to optimal, but slightly better performance can be had on a per-system per-distribution-method basis. This information is hard-coded into the qarray implementation. Should new architectures have different performance characteristics, that information can also be added.

The size and alignment of elements within an array can be critical to taking full advantage of the cache as well as avoiding unnecessary bus traffic. Caches almost always load aligned data from memory. Accessing unaligned data can result in multiple load instructions, slow compositing instructions, and even crashes on some architectures. For example, Figure 5.6(c) shows that the penalty for using unaligned data can be as high as a 50% reduction in bandwidth. The spikes and variances in bandwidth shown are consistent over multiple tests, not the result of temporary issues. Avoiding these situations and aligning data when beneficial is a task that is often left to the compiler to handle, particularly for pre-allocated global and stack-based variables.

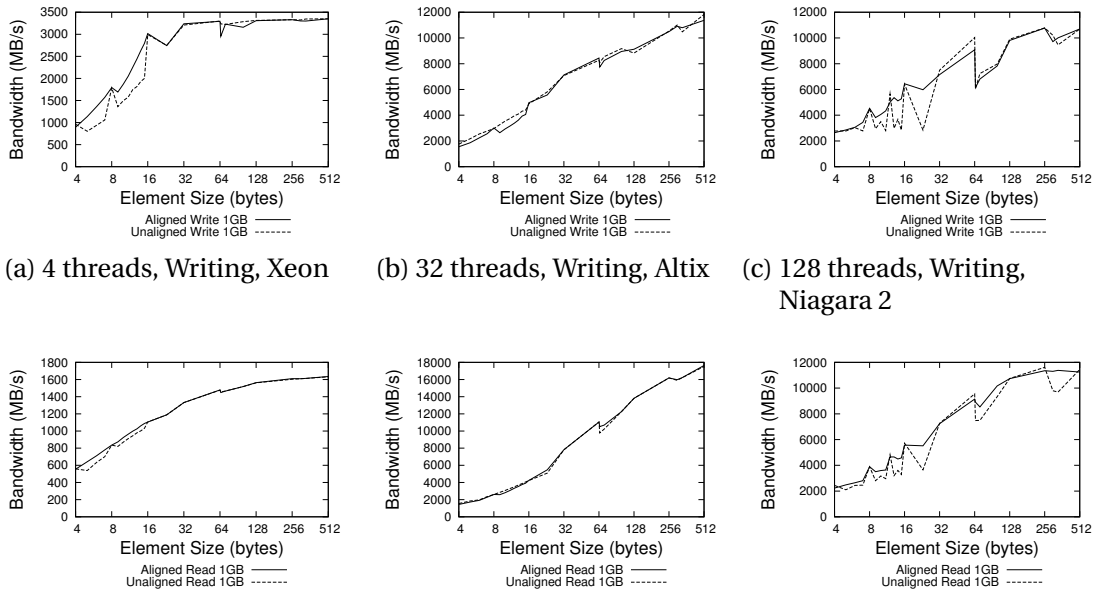
Compilers often pad data structures to ensure that they can be easily aligned in arrays. Since a qarray performs layout at runtime rather than at compile-time, data alignment must be handled manually.

Figure 5.6 presents average results from the second benchmark, which demonstrates the impact of element size and alignment on performance. These graphs compare the memory bandwidth achieved while reading and writing data to arrays with a variety of element sizes. Specifically, an "unaligned" array whose elements are not padded is compared to one that enforces an 8-byte alignment by adding padding to round element size to the next-largest multiple of eight.

On all three systems, manually aligning data has a clear benefit for small element sizes, but has less of an impact on performance when using large elements. This is likely because unaligned layout is more condensed, and that benefit outweighs the penalty for unaligned access beyond a certain size element. Large elements can also amortize the alignment penalties, depending on their internal structure. This information is also coded into the qarray implementation, which automatically rounds element sizes under 64 bytes to the next-largest multiple of eight. This padding can be prevented, if compact data layout is desired, by using the `qarray_create_tight()` function when creating the array.

5.3.3 Distributed Queue

Queues have a spectrum of uses in parallel applications, and optimal queue design depends heavily on the intended use. One common use for a queue is as a buffer between two threads. In that case, absolute ordering is required,



(a) 4 threads, Writing, Xeon (b) 32 threads, Writing, Altix (c) 128 threads, Writing, Niagara 2

(d) 4 threads, Reading, Xeon (e) 32 threads, Reading, Altix (f) 128 threads, Reading, Niagara 2

Figure 5.6: Element Size/Alignment Scaling

These graphs compare the memory bandwidth achieved by multiple threads iterating over one million element arrays using both 8-byte aligned data and unaligned compact data of various element sizes. Alignment provides a speed improvement for small element sizes, but is less important for larger elements. By default, the qarray rounds element sizes below 64 bytes to the next-highest multiple of eight.

but because there is only one producer and one consumer, assumptions may be made in the implementation to provide additional speed. For example, there will be low contention for the head and tail of the queue, so there need only be one of each. Queues are also used for distributing work among multiple threads, organizing multiple producers and multiple consumers. In that situation, the guarantees required of a simpler queue—such as absolute temporal ordering of all items added to the queue—may be relaxed in order to decouple portions of the queue and thereby speed operation.

Locality has greater importance for the latter queue use. A locality-aware distributed queue prefers consuming queue elements from nearby producers over consuming the oldest elements in the overall queue. This is a useful policy for the implementation of work-stealing task queues, for example, because it assists in preserving locality of reference between tasks by preferentially executing them near where they were created.

5.3.3.1 Design

The qthread library provides two types of queue. The queue that guarantees global first-in-first-out ordering is a lock-free queue called "qlfqueue," based on Michael and Scott's queue [117]. The distributed queue, or "qdqueue," inspired by the "stochastic distributed queue" by Johnson [91] and the push-based queue by Arpaci-Dusseau [11], only guarantees end-to-end ordering. The two queues provide nearly identical interfaces, with different prefixes. New lock-free queues are created with the `qlfqueue_create()` function and destroyed with the `qlfqueue_destroy()` function. New distributed queues are created with `qdqueue_create()` and destroyed with `qdqueue_destroy()`. Elements may be queued with the `qlfqueue_enqueue()` or `qdqueue_enqueue()` functions, and dequeued with the `qlfqueue_dequeue()` or `qdqueue_dequeue()` functions. The queues may also be quickly checked for queued elements using the `qlfqueue_empty()`/`qdqueue_empty()` functions.

The core task of any end-to-end ordered distributed queue is matching consumers to producers. A centralized matchmaker that tracks the location of all queued elements is a simple approach, but one that creates a bottleneck both logically and in communication patterns. This design can be mod-

ified to distribute matchmaking via hierarchical matchmaking deputies that are responsible for clusters of nodes, and elevate match requests that cannot be satisfied locally to the global matchmaker. This distribution reduces the contention of consumers, but increases the work of producers without addressing their bottleneck issues. An alternative approach is one known as a "stochastic distributed queue" [91], where consumers repeatedly probe producer queues until a non-empty queue is found. This technique is particularly efficient when the queues are rarely empty. However, when consumers are frequently in search of non-empty queues, the stochastic approach requires a great deal of work for consumers and creates multiple bottlenecks as each consumer polls the entire set of producers. This work can be reduced if consumers and producers cooperate—via advertisements, somewhat like a push-based queue [11]—without significantly impacting the common-case operations when empty queues are rare.

The qdqueue uses a separate qlfqueue for each location in the system. Each location also maintains a list of "advertisements" received, sorted by distance to the advertiser, and a record of the last consumed element's source. New elements are enqueued in the local queue. If the queue was empty, any waiting local consumers are notified. If the queue was not empty, the enqueuer posts "advertisements" to nearby queues. An advertisement informs remote consumers that there is data available in this location. The set of "neighbor" locations to receive advertisements is determined at setup time, based on topology. Thus, the maximum work of a producer is fixed, independent of the total size of the system. A fast producer can avoid resending advertisements by tagging them with a monotonically increasing counter and tracking whether

advertisements have been consumed. If the last-issued advertisements have not been consumed, no additional advertisements are necessary.

To dequeue an element, the local queue is checked first. If the local queue is not empty, an element is dequeued. After successful dequeueing, the consumer records that the last-dequeued element was in the local queue. If the queue was empty, any advertisements that had been received are used to assist in finding non-empty queues, and are checked in order of distance from the consumer. When responding to an advertisement, it is useful to update the advertiser's record of advertisements consumed. If an element is dequeued from a remote queue, the location of that queue is recorded as the source of the last-dequeued element. When checking empty remote queues, if that queue's last-consumed record points to another location it is considered to be an advertisement to check. If an advertisement is received while checking remote queues, it is checked before any other remote queues. Thus, consumers in search of elements to dequeue can cooperate in locating non-empty queues. If none of the advertisements result in a found element, it is necessary to check *all* remote queues, in the order of distance from the consumer, using the same procedure as responding to advertisements. If all remote queues are empty, the consumer must either return empty-handed or wait for an ad to be posted. This distributed data structure assures that queued data is preferentially consumed near its production and that end-to-end ordering from one producer to one consumer is maintained.

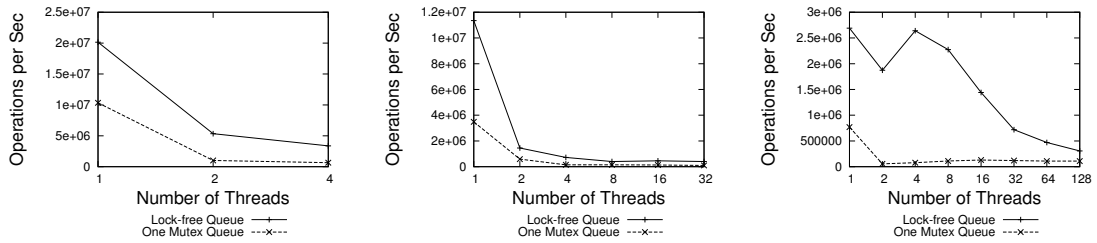
5.3.3.2 Benchmark

To illustrate the different performance characteristics of the two types of queue, four simple benchmarks were designed. They each use two threads per location, one to enqueue 4-byte elements into a shared queue, the other to dequeue them. The queue is shared among all of the threads. The producer threads enqueue 10,000 elements, and the consumer threads dequeue 10,000 elements before exiting.

The first benchmark uses the lock-free qlfqueue as the shared queue, the second uses pthreads and shares a cprops-based mutex-protected queue. The third benchmark is identical to the first, except it uses the distributed qdqueue as the shared queue. The fourth benchmark is designed to be very similar to the other benchmarks, but uses Intel Threading Building Blocks (TBB) to create threads using a `parallel_for` operation, and uses the TBB's own `concurrent_queue` as the shared queue. Both the third and fourth benchmark can also use large 1024-byte queue elements rather than 4-byte elements.

5.3.3.3 Performance

Figure 5.7 illustrates the effect of strict ordering on the scalability of a queue by comparing the lock-free qlfqueue to a single-mutex queue implementation from the cprops library [2]. Operations per second were calculated by taking the number of operations performed (10,000 enqueue operations and 10,000 dequeue operations per location) and dividing by the time necessary to finish the entire benchmark. The first and second benchmarks were both run ten times on each test system, with each level of parallelism. Figure 5.7 presents



(a) Xeon (b) Altix (c) Niagara 2

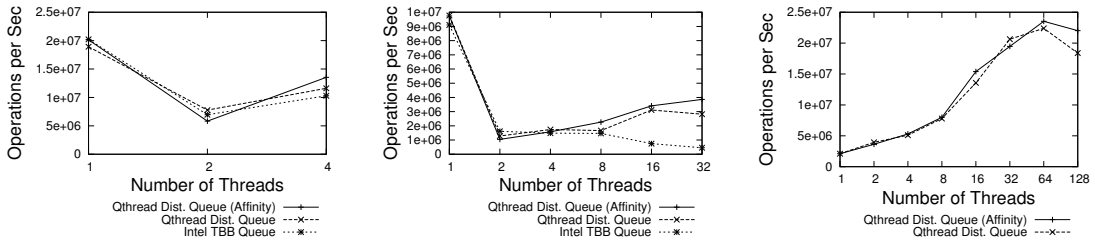
Figure 5.7: Scaling Simple Ordered Queues

These graphs compare the enqueue/dequeue operations per second achieved by strictly ordered queues over different numbers of producers and consumers. Lock-free queues are significantly faster than mutex-based queues, but serialization to ensure ordering limits the scalability of such queues.

the average results of those ten runs; each data point is the average result of ten benchmark runs.

Both the mutex-based cprops queue and the lock-free qlfqueue queue ensure global ordering via a serialized critical section: the serialization defines the ordering. Using a lock-free queue can be orders of magnitude faster than a mutex-based queue because it uses hardware assistance to minimize the size of the critical section as well as the overhead of synchronization. Nevertheless, this serialization acts as a bottleneck, preventing strictly ordered queues from scaling. Additional threads, rather than increasing the number of operations performed per second, increase the contention for the critical section, and decrease the number of operations that can be performed per second.

Figure 5.8 compares the performance and scalability of the qdqueue and the Intel Threading Building Blocks `concurrent_queue`, which provide a similar end-to-end-only ordering guarantee. The qthread distributed queue can operate in two modes, one where its queues are strictly tied to processing nodes, and one where they aren't. The stricter mode is labeled in the graphs as "Affin-



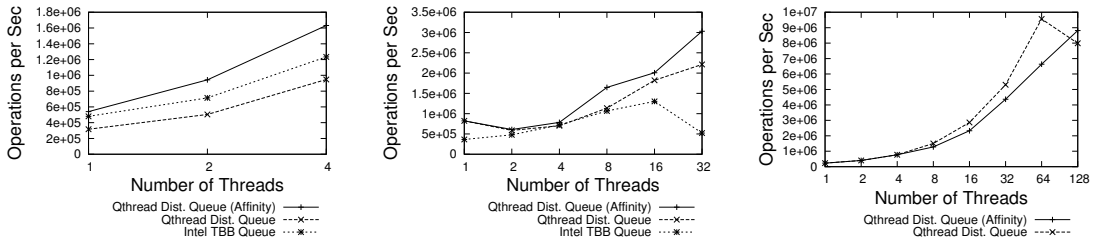
(a) Xeon (b) Altix (c) Niagara 2

Figure 5.8: Scaling Distributed Queues

These graphs compare the enqueue/dequeue operations per second of a locality-aware distributed queue to the Intel Threading Building Blocks `concurrent_queue`. *Locality-awareness provides significant benefits in complex systems.*

ity" mode, and has a performance edge on the more complex Altix machine (Figure 5.8(b)).

The overhead of the distributed queue's locality-aware design does not significantly impact single-threaded performance; both the distributed queue and the lock-free queue (Figure 5.7) provided similar speed with a single thread on all three systems. However, the extra logic and relaxed ordering requirements of the distributed queue provide significant performance improvements over the lock-free queue. Most fundamentally, with two or more threads, operations per second increase rather than decrease when additional threads are used. In more concrete terms, at scale, the distributed queue performed 9.6 times better than the lock-free queue on the Altix, 72 times better on the Niagara 2 server, and 4 times better on the Xeon workstation. The Niagara 2 server provided the most impressive performance improvements because most threads share access to a cache, which minimizes the cost of inter-thread communication. The Altix and Xeon have aggressive cache-coherency protocols, which



(a) Xeon (b) Altix (c) Niagara 2

Figure 5.9: Scaling Data-Laden Distributed Queues

These graphs compare the enqueue/dequeue operations per second of a locality-aware distributed queue to the Intel Threading Building Blocks `concurrent_queue` *when passing large amounts (1024-bytes) of data. Locality-awareness is particularly important when handling large blocks of data, because there is a high penalty for non-local dequeueing.*

penalize any shared information. Thus, even the minimal cooperation between consumers reduces throughput versus single-threaded operation.

The benchmark in Figure 5.8, however, is only transferring small word-size pieces of data. When the queued data is large, the impact of cache coherency is reduced and the importance of locality becomes clearer. Figure 5.9 illustrates the effect of using end-to-end ordered distributed queues to handle larger (1024-byte) blocks of data.

The larger the data being queued, the higher the cost of transferring the data to another node, and thus the greater the penalty for ignoring the locality of data and computation. Large blocks of data also tend to keep queue management information out of the cache, and thus cache coherency is not as significant a problem.

5.4 Conclusion

Developing portable locality-aware data structures, even with the advantage of a threading library with an integrated locality framework is a significant challenge. This chapter presented several data structure designs that use locality information to adapt to system topology at runtime and examined the selection of optimal system-specific design parameters. The effectiveness of locality-aware design in distributed data structures was demonstrated. Memory pools can be up to 155 times faster than traditional `malloc()` while providing location-specific memory. The qarray distributed array design supports strong-scaling on large ccNUMA systems, executing 31.2 times faster with 32 nodes than with one. The qdqueue distributed queue design provides up to a 47 times improvement over a fast lock-free queue by providing only an end-to-end ordering guarantee. Its locality-awareness provides a benefit of up to 830% improvement in performance over state-of-the-art concurrent queues on a large ccNUMA system. Importantly, the use of locality information does not significantly impact serial performance or performance on small systems with uniform memory access latencies.

CHAPTER 6

ADAPTIVE COMPUTATIONAL TEMPLATES

6.1 Introduction

The previous chapter presented three locality-aware data structures that alter their behavior — both communication and data placement — based on the topology of the underlying computer system at runtime. In so doing, as discussed in Chapter 4, they carefully map their underlying structure to the hardware resources to take better advantage of that hardware. This would be more difficult without having locality integrated into the threading interface.

This chapter presents the design of three computational abstractions based on the qthread API that adapt to machine topology without the programmer needing to explicitly consider memory topology and demonstrates their performance on the same range of machine topologies used in the previous chapter. The abstraction are sorting, all-pairs, and wavefront. Several methods of choosing computation locations based on input location are also examined.

Developing large-scale multithreaded applications is quite difficult, simply because there are many events happening at the same time, which results in potential errors and performance tuning issues that do not affect serial programs. As discussed in Chapter 4, these issues include data structure contention, deadlock, race conditions, communication overhead, and more. Poor

choices in application design can prevent the program from taking full advantage of available hardware resources. However, coordinating multiple threads of execution while keeping all potential issues in mind and tailoring the design to a generic hardware topology requires the skill of a distributed computing expert.

Due to the specialized skills required to build high performance parallel software from scratch, it has become commonplace to use additional programming abstractions to simplify the programming problem. Such abstractions provide a simple interface for the programmer with clear rules that are easy to reason about while hiding the details of how the system will realize the abstraction.

The challenge of a good computational abstraction is to hide the details of the underlying system and adapt to whatever system is available. It must provide a natural-seeming means of segmenting the necessary work into parallel tasks, and must then have a way of deciding where and when to perform each parallel task.

One easy metric to use when evaluating a computational abstraction is to see how transparently it can avoid basic problems like false sharing [53] and minimize the necessary communication. As a simple example, the simplest map function over a distributed data structure is a single thread that can iterate over the distributed data structure. If the thread can migrate, every migration is communication, so an implementation that executes the mapping function on every element in the data structure in a given location before moving on will have less communication than one that uses a random or linear iteration scheme. As another example, in many cases each operation that needs

to be performed has more than one input and produces an output. Thus, the best location to perform each operation depends on the locations of its inputs and outputs; the optimal location minimizes the overhead for all of the necessary communications.

6.2 Sorting

Sorting is a somewhat unusual computational abstraction. As a generic function, it is used in many applications to create sorted lists or to group categories of data. But in concept, the "sort" simply provides a framework for applying a "comparison" function to all pairs of data in a group, eliminating duplicate work by assuming that "comparisons" are transitive. The comparison function may do other things, such as return random values or modify the things being compared, to achieve a non-traditional result. As such, sorting algorithms can be used as shuffle algorithms, routing algorithms, graphical posterization algorithms, and normalization algorithms, among other things.

Because it is such a popular and convenient tool, sort has been well-studied. Sorting in new architectures, however, presents some unique challenges and some unique opportunities. For example, while parallel sorting algorithms are easy to imagine — merge sort is a sort with clear parallelism opportunities — they do not always work well. The merge sort is a two-stage recursive sorting algorithm: first divide the data to be sorted in half (recursively, until they are sorted), then merge the two halves together. While splitting data arbitrarily is easy to do in parallel, the merging phase is difficult to parallelize without specialized hardware; most implementations rely on the existence of PRAM [32, 107]. Even more recent parallel implementations of merge sort [89]

cannot do better than 2.3x improvement on eight commodity processors. Another popular sorting algorithm is the QuickSort, which provides impressive performance in serial applications. In any QuickSort algorithm, there are also two phases: partitioning the data into two segments around a "pivot" point, and then sorting each segment independently. The key to parallelizing the QuickSort algorithm is a parallel partition algorithm.

6.2.1 A Parallel Partition Algorithm

The basic Hoare QuickSort partition algorithm [34] is given in Figure 6.1. In order to parallelize this algorithm, the input must be divided among multiple threads. Each thread can partition its subset of the input around the specified pivot. If each thread's portion of the input is entirely separate from that of the other threads, the partitioned portions will need to be merged after all of the threads have finished partitioning. By interleaving the thread divisions, this merge phase can be avoided.

A quick way of dividing an array among n threads is to make each thread responsible for every n'th array element. Figure 6.2 illustrates this idea using a small ten-element array with two threads. At the top of the figure is the initial, unsorted array. In the next line of the figure, the array has been logically divided between two threads; the slightly raised numbers are assigned to thread one, and the slightly lowered ones are assigned to thread two. The pivot is chosen to be 4.4. A variation of the standard QuickSort partitioning algorithm can then be applied by both threads to their subsets of the array, resulting in the third line of the figure. The only difference is that partition function must skip the parts of the array that belong to other threads. The arrows here represent

```
1:  procedure Hoare-Partition(A, p, r)
2:      pivot ← A[p]
3:      i ← p − 1
4:      j ← r + 1
5:      while TRUE do
6:          repeat
7:              j ← j − 1
8:          until A[j] ≤ pivot
9:          repeat
10:             i ← i + 1
11:         until A[i] ≥ pivot
12:         if i < j then
13:             exchange A[i] ↔ A[j]
14:         else
15:             return j
16:         end if
17:     end while
18: end procedure
```

Figure 6.1. Hoare QuickSort Partition Algorithm

the left (darker) and right (lighter) iterating edges of each thread's partitioning algorithm, respectively, the i and j values in Figure 6.1. The threads then cooperate to calculate the maximum of their right edges and the minimum of their left edges to define a "fuzz" range. Outside this "fuzz" range, the array is guaranteed to be properly partitioned around the pivot. This fuzz range may then have the same parallel partitioning scheme applied to it, or if it is sufficiently small it may be quickly partitioned serially.

Such a simple partitioning scheme works well on the Cray XMT. Memory addresses in the XMT are distributed throughout the machine at word boundaries. When dividing work amongst several threads on the XMT, the work regions can be as fine-grained as a single word without significant loss of

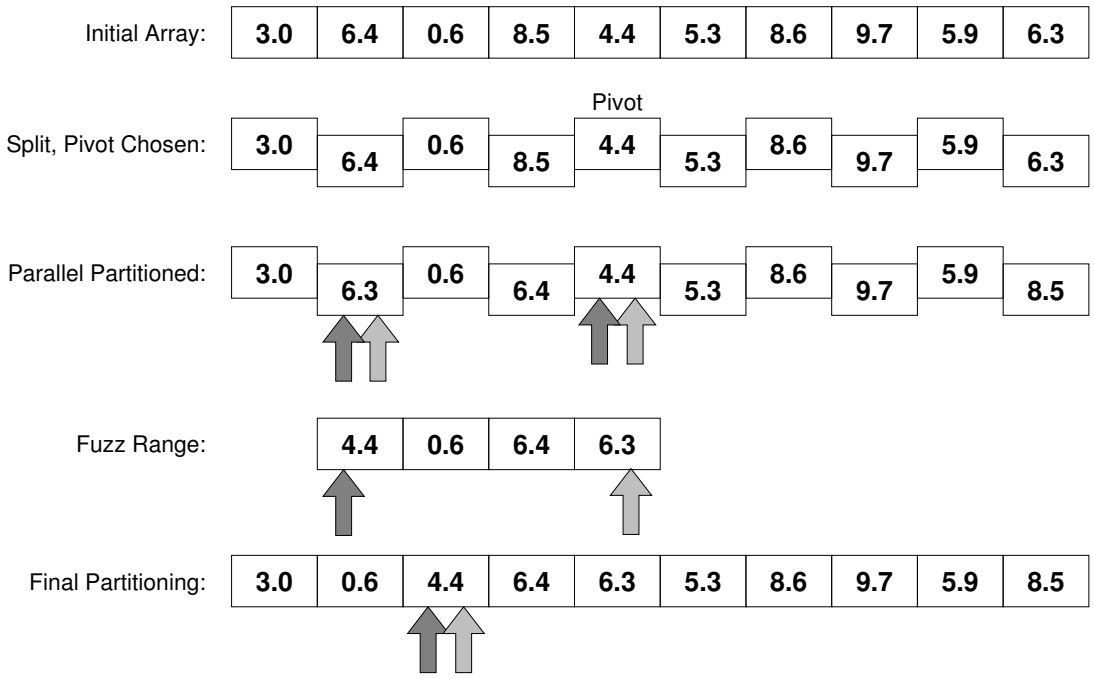


Figure 6.2. Basic Parallel Partition Scheme

performance. Conventional processors, on the other hand, are optimized for the common-case situation where memory is accessed linearly. They assume that memory within page boundaries is contiguous and therefore use caches with large cache lines. By optimizing for linear access with large 32-, 64-, and even 128-byte cache lines, conventional cache designs limit the granularity at which tasks can be divided among multiple processors without paying a heavy cache coherency penalty. Thus, on conventional processors, this basic parallel partitioning scheme is very likely to result in poor performance. Consider that floating point numbers often use 8 bytes of memory while cache lines in modern commodity processors are usually somewhere between 32 and 128 bytes, or enough to contain between four and sixteen array elements. If this simplistic partitioning algorithm is used, threads on different processors

attempting to modify their independent portions of the array will be forced to invoke the cache coherency protocol because their independent portions use the same cache lines. Constantly sending the same memory back and forth between processors to resolve cache coherency prevents the parallel algorithm from exploiting the capabilities of multiple processors. Using additional processors may even result in slower performance than fewer processors. Clearly, the hardware must be taken into consideration when designing an implementation of a computational abstraction.

The qthread library includes an implementation of the QuickSort algorithm that avoids contention problems by altering the granularity of data sharing to match the current system's cache line size. The idea is illustrated in Figure 6.3. The array being sorted is the same as the previous figure, but instead of dividing the array on single-element boundaries, elements are distributed among the two threads in groups of four. This can create a slight load imbalance between the partitioning threads — in this example, thread one must partition six numbers and thread two only partitions four — but this is an insignificant difference when sorting large arrays and is never larger than a single cache line. Again, the two threads partition their portion of the array, and again, a fuzz region must be handled to reconcile the entire partitioning.

6.2.2 Performance

Grouping array elements along cache-line boundaries avoids cache-line competition between processors while still exploiting the parallel computational power of all available processors on sufficiently large arrays. This modification, while small, is significant.

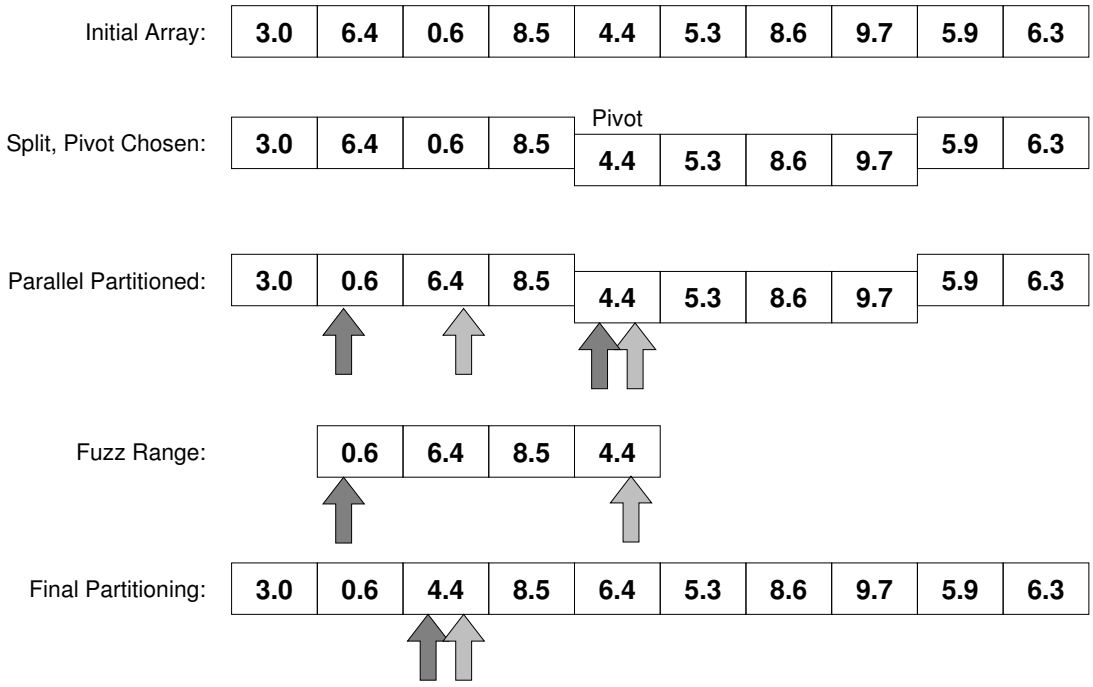


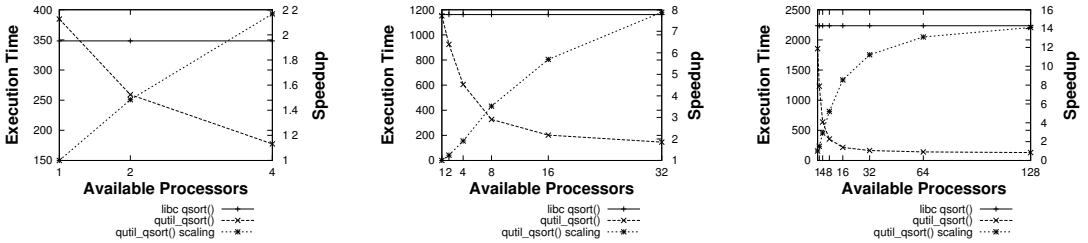
Figure 6.3. Cache-line Aware Parallel Partitioning

6.2.2.1 Benchmark

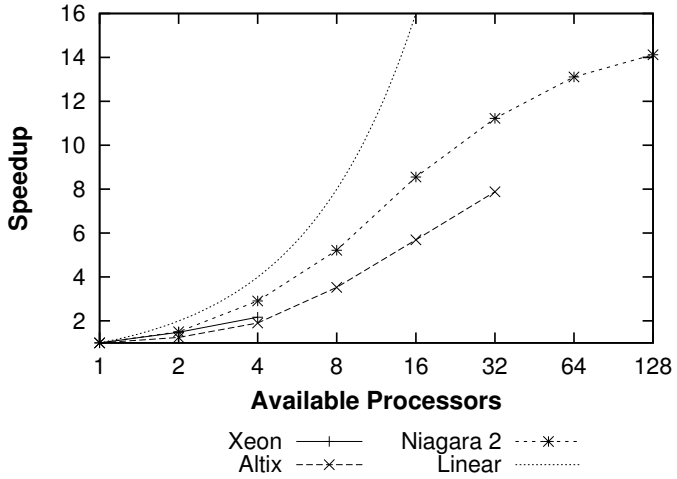
A benchmark was designed to illustrate the scalability of the qthread-based QuickSort implementation. The benchmark creates an array of one billion double-precision floating point numbers, initializes them with random values, and then sorts them. This array occupies approximately 7.4 GB of memory. For comparison, a similar, trivial benchmark sorts the same array with the libc `qsort()` function.

6.2.2.2 Results

Figure 6.4 illustrates the scalability of the sorting benchmarks. Both benchmark sort the array ten times on the three architectures discussed in the previ-



(a) Xeon (b) Altix (c) Niagara 2



(d) Scaling `qutil_qsort()`

Figure 6.4. Libc's `qsort()` and `qutil_qsort()` Sorting 1 Billion Floating Point Numbers

ous chapter. The figure presents the average execution times. In Figures 6.4(a)–(c), the execution times of the benchmarks are directly compared. The speedup is also provided in these graphs, on the right-hand y-axis, as well as in Figure 6.4(d).

Figures 6.4(a)–(c) illustrate that on every architecture only two threads are necessary to provide a faster sort than the standard `qsort()`. While linear scaling is not achieved, because in a random list there is no guarantee of either locality or uniform load, nevertheless the algorithm does provide solid per-

formance improvement with additional hardware. The 128-thread sort on the Niagara 2 is over 14 times faster than single-threaded sorting, a 32-thread sort on the Altix is over 7 times faster than single-threaded sorting, and a 4-thread sort on the Xeon is just over 2 times faster than a single-threaded sort. It is interesting to note that the libc `qsort()` implementation on Solaris is not especially good; the single-threaded version of the parallel QuickSort is faster.

6.3 All-Pairs

The All-Pairs abstraction is a computational abstraction applicable to a wide range of problem categories in several scientific fields. Fundamentally, the All-Pairs abstraction takes as input two sets of data and a "combination" function that accepts two elements as input, one from each data set. This function is then applied to all pairs of elements from the two sets. Mathematically, with one set of n elements and another of m elements, the combination function must be called $n \times m$ times. If the combination function is commutative, the number of times the function must execute can be reduced to $\frac{n!}{(n-2)!} + n$. The basic concept is illustrated in Figure 6.5, where elements from set A and set B must be combined.

All-Pairs is useful in a wide range of problem categories from graph theory to biometrics to data mining. In graph theory, the all-pairs shortest path problem is well-studied, and used for all manner of routing problems. The field of biometrics often finds itself computing all-pairs issues when testing new algorithms. Because biometrics is the study of measuring and identifying human biological characteristics, one of the more common questions that is asked is whether one sample is from the same source as another sample. Frequently,

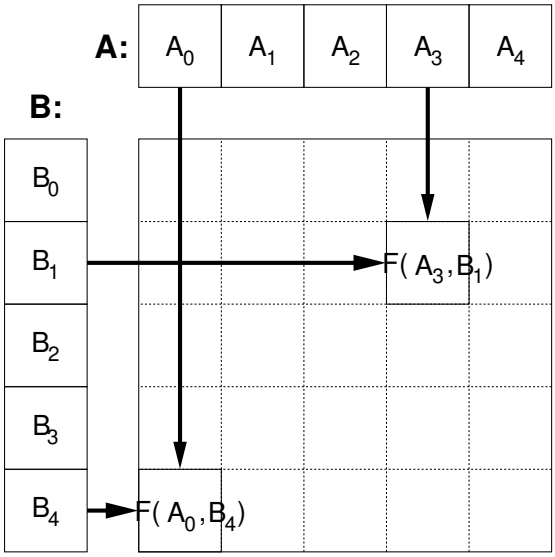


Figure 6.5. The All-Pairs Problem

a large database of samples from known sources is built and new identification algorithms are tested by using them to compare every sample to every other sample. The algorithms produce some sort of output, and a collection of this output from the comparison of every sample to every other sample is a “similarity matrix”, which represents the accuracy of the comparison function. This matrix can be compared to the matrix of other algorithms, enabling quantitative analysis of algorithm effectiveness.

In data mining applications, the behavior is similar. One phase of many data mining knowledge discovery algorithms is to react to bias or noise within a block of data. Different classification algorithms are effective in combating different types of noise in data. A common way of testing classification algorithm effectiveness is to run the algorithm against many different examples of noise combined with many different examples of data. A large database of

noise categories (defined as distributions) and example data (also defined as distributions) is built, and a test harness function is used to combine noise with data and feed the result to the classifier being tested.

It is worth pointing out that not all problems that can be phrased as an All-Pairs problem are most efficiently computed that way. For example, searching a block of data for a given value can be presented in terms of an All-Pairs problem: compare the search-key (a set with one element) with every element in another, much larger, set. Such problems can usually be better served by more specific algorithms. What is being considered here is exclusively problems which actually require *all* of the output values from executing the combination function with every pairing of data from the two input sets.

6.3.1 Design

One of the convenient aspects of the All-Pairs problem is that every comparison can be computed independently, which makes job scheduling somewhat simpler. When generating an output matrix in a parallel setting, the key issue then becomes data distribution: where are the inputs, and based on that, where should the comparison function execute? Viewed as a ThreadScope graph where every cell's computation is performed by a separate thread, as in Figure 6.6, it is unclear that there is a good way of dividing the problem into localized pieces. The input is distributed among threads quite broadly.

To address the question of discovering input's location, this All-Pairs implementation takes input data sets in the form of qarrays. Each element of a qarray has an inherent location, and those elements are even organized in groups that are then distributed across the system. As such, the basic inter-

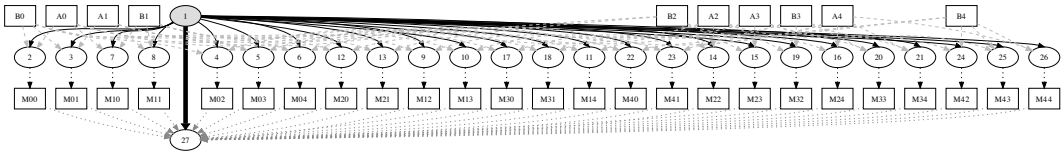


Figure 6.6. The Generic All-Pairs Structure, 5×5 Input

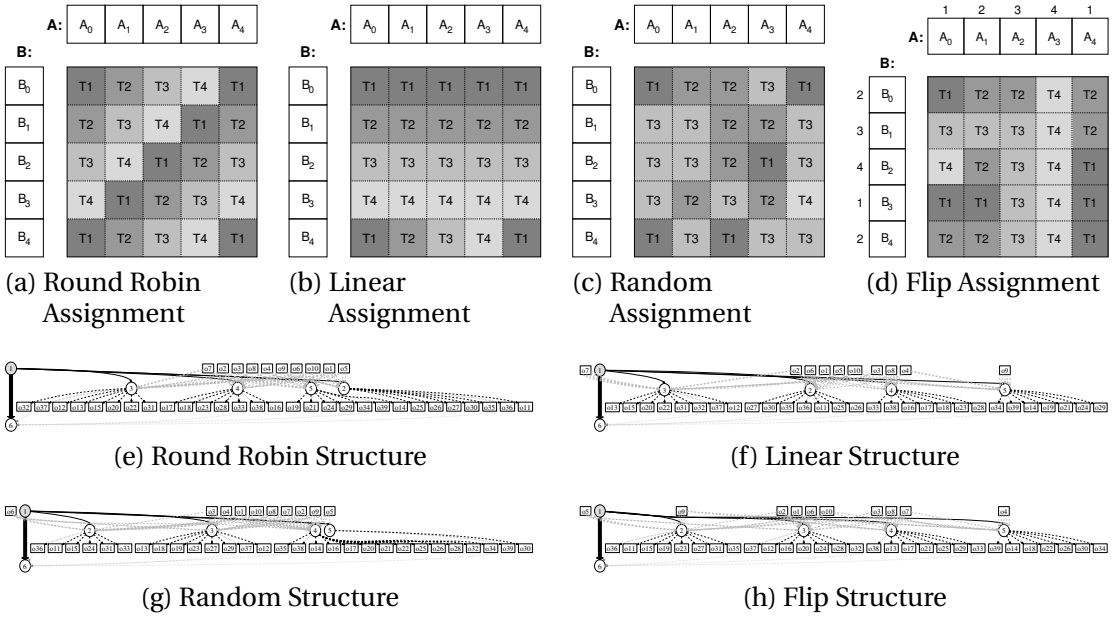
```
typedef void (*dist_f) (const void *unit1, const void *unit2);

void qt_allpairs(const qarray *array1, const qarray *array2,
                 const dist_f distfunc);
```

Figure 6.7. The Qthread Library's All-Pairs Interface

face to the All-Pairs computational abstraction is the function `qt_allpairs()`, as defined by the function prototypes in Figure 6.7. In that prototype, `array1` and `array2` are the input data sets and `distfunc` is a function for combining or comparing elements from both arrays.

The next design issue is the question of scheduling work: how, and where? This All-Pairs implementation takes the worker-thread approach, creating worker threads for each location in the system. These worker threads fetch work units from a distributed qdqueue, thereby prioritizing nearby work over more distant work. Because each work unit can be executed independently, work units can be inserted into the work queue in any order, and the queue can essentially be pre-loaded with work. Thus, a separate thread is responsible for adding work units to the queue.



(a) Round Robin Assignment

(b) Linear Assignment

(c) Random Assignment

(d) Flip Assignment

(e) Round Robin Structure

(f) Linear Structure

(g) Random Structure

(h) Flip Structure

Figure 6.8. Impact of Distribution on All-Pairs Structure

While work units can be added to the distributed queue without paying attention to the location, work units will execute faster in some locations than they will in others based on the proximity of their input.

Figure 6.8 illustrates the difference that assignment of tasks to worker threads can have on the structure—and thus the ease of mapping to hardware—using ThreadScope structure graphs that have the distributed queue removed (thus showing only the result of the assignment). In each case, the outputs (and their associated inputs) are assigned to each of four threads statically. There are five elements in each of the two input sets, and they are each considered separately. The assignments are illustrated above the structure graphs.

One of the convenient aspects of using GraphViz to generate ThreadScope visualizations is that memory objects tend to cluster near the thread objects

that access them the most. Thus, in these structure graphs, the extent to which the assignments cause input elements to cluster around each of the four worker threads illustrates how closely the inputs and worker threads are associated. The three assignment techniques are fairly straightforward. The "round robin" method, Figure 6.8(a), assigns each element of the output matrix to each worker thread in turn. The "linear" method, Figure 6.8(b), assigns output matrix computation to worker threads according to the associated input from the B set. The "random" method, Figure 6.8(c), assigns elements randomly (the graphed example is only one of the possible assignments). And the "flip" method, Figure 6.8(d), first assigns each input to a thread, and then assigns each output element to one of the two input threads, randomly. In the ThreadScope graphs of each assignment pattern, Figures 6.8(e)–(h), the inputs are the square blocks along the top of each graph. The extent to which those inputs cluster together or separate themselves indicates how much those inputs are associated with a single worker thread. The example with the best cluster separation is the "flip" assignment example. The mapping of inputs to worker threads affects the amount of communication necessary to complete the task, and as such has the potential to affect performance.

6.3.2 Performance

Given two equally weighted inputs with known locations, the optimal place to process both inputs is a location that is close to both inputs. Several options are considered here:

All Same Enqueue all work units in location zero and simply have all work threads pull from that single queue. This is used as a baseline.

Random Enqueue work units in random locations, regardless of their inputs.

Flip Enqueue each work unit in the same location as one of its inputs, chosen randomly.

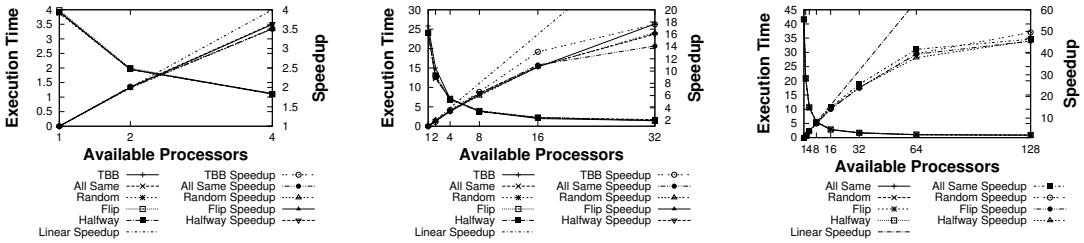
Halfway Prefer to execute each work unit in the location where the sum of the distances to both inputs is minimized, compared to other locations.

6.3.2.1 Benchmark

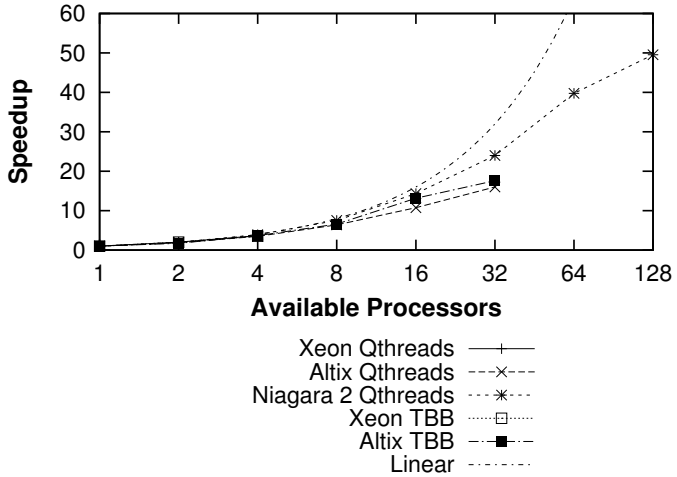
To demonstrate the scalability of the All-Pairs framework, and to evaluate the design decisions, a benchmark was created. Two 30,000-element qarrays using the static hash distribution are created and initialized randomly. These qarrays and a function to multiply two input numbers and store the output in a specified location are then given as arguments to the All-Pairs function. The products are stored in a large (approximately 6 GB) output matrix. Each placement algorithm was used with this algorithm ten times on each of the three machines discussed in the previous chapter. It was compared to a similar implementation using the Intel Threading Building Blocks (TBB). Figure 6.9 presents results of that benchmark.

6.3.2.2 Results

The framework scales nicely on all three architectures, and has comparable performance to the TBB implementation, but is more portable. There is surprisingly little performance difference between the various methods of choosing where to enqueue work, and surprisingly little difference between the qthread-based implementation and the TBB implementation.



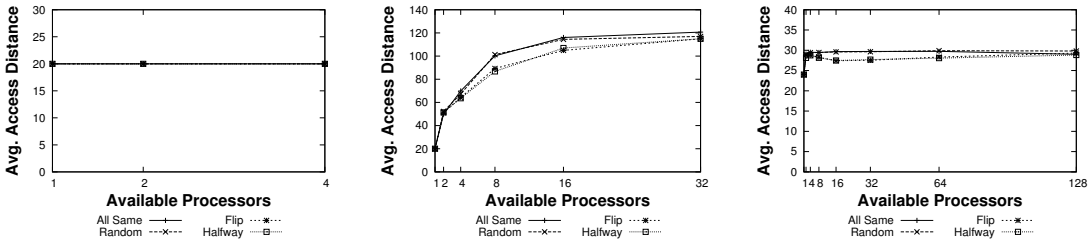
(a) Xeon (b) Altix (c) Niagara 2



(d) Scaling `qt_allpairs()`

Figure 6.9. All-Pairs, 30,000×30,000 Pairs with Several Distribution Methods

One plausible explanation for the lack of variance in performance between the different placement heuristics is that in this benchmark, access to the inputs is not a limiting factor in performance, which suggests that if the combination function was more complex or accessed its inputs more frequently the performance characteristics might be different. To further explain the behavior, the benchmark was instrumented to record the distances to the inputs of every work unit that was completed. The average distances to the inputs for each work unit gives a good rough estimate of the effectiveness of each work-



(a) Xeon Distances (b) Altix Distances (c) Niagara 2 Distances

Figure 6.10. All-Pairs Work Unit Placement Heuristics

unit placement heuristic. Figure 6.10 illustrates these distance averages. Because there is so little variance in the distances to input memory across the various distribution heuristics, there is naturally very little performance difference.

6.4 Wavefront

The Wavefront abstraction is a more complex computational abstraction than either sorting or All-Pairs. It defines a two-dimensional recurrence relationship within its output data. Like the All-Pairs abstraction, it takes two ordered data sets as input and creates an output matrix. However, whereas all elements of the output matrix could be computed independently in the All-Pairs abstraction, in a Wavefront computation each output value depends on the output values one row down, one column over, and both one row and one column over. In other words, in the output matrix M, the value of $M[i, j]$ is the result of some function $F(M[i-1, j], M[i, j-1], M[i-1, j-1])$. The two ordered data sets taken as input to the function are considered to be the first row and

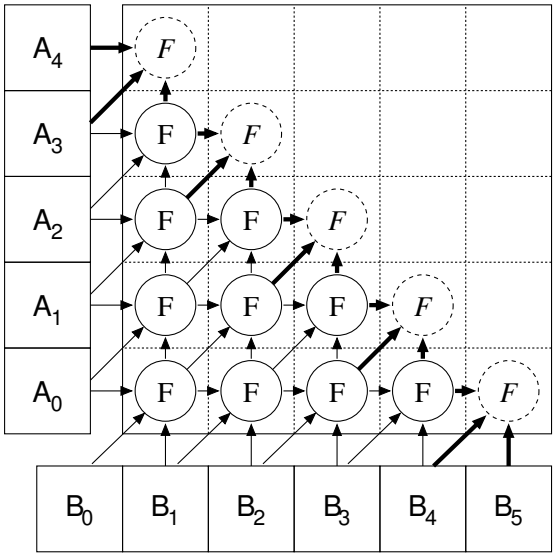


Figure 6.11. The Wavefront Abstraction

first column used in generating the output matrix. The relationship is illustrated in Figure 6.11.

The Wavefront abstraction has a rich history in scientific computing. It was first described as a "hyperplane" method by Lamport [103], and has proven to be useful in a range of simulations problems, from economics and game theory to particle physics [97] and parallel solution of triangular systems of linear equations [74, 146, 177] to genetic sequencing. For example, in game theory, the input can be all possible ending states of a game, and the recurrence relation can be used to work backwards to discover the path of decisions that results in each outcome. In genetic sequencing, a Wavefront recurrence relation is used to describe—via dynamic programming—the sequence alignment problem. The Smith-Waterman algorithm [161], for producing lo-

cal alignments, is a typical example of a wavefront-based gene sequencing algorithm.

6.4.1 Design

One of the more challenging aspects of Wavefront-style problems is the dependence of successive data calculations. Previous wavefront implementations typically rely on custom compiler optimizations and partial pipelining to improve performance while guaranteeing that data will be ready when it is needed. A naïve threaded implementation of a wavefront framework might use a separate thread for each output matrix element that must be computed, trusting on synchronization mechanisms—such as full/empty bits or atomically incremented progress counters—to ensure that threads become runnable only when their inputs are ready. The ThreadScope structure of a small version of such a program is illustrated in Figure 6.12. Such an architecture would, of course, work, but would be wasteful: each thread requires state, and allocating a thread's worth of state for every element in a large output array can quickly run out of state. While it is certainly possible to use a separate thread for each output value, however, that is almost certainly a wasteful design.

Much like the All-Pairs abstraction, one of the key aspects of an efficient Wavefront design is locality. In the All-Pairs abstraction, the locality of each calculation was defined by its input, and as such, the input was in the form of qarrays, making the location of any given piece of data easy to determine. The Wavefront abstraction uses a similar approach for input, as illustrated by the interface, defined in Figure 6.13. However, in the Wavefront abstraction,

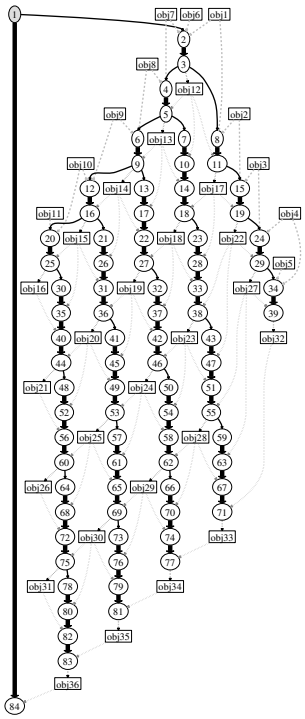


Figure 6.12. Naïve Wavefront ThreadScope Structure, 5×5 Input

the output of each calculation becomes input to the next few steps; thus, the location of the output is important to track as well.

This Wavefront abstraction makes the assumption that the final edges of the output matrix are the most important. Based on that assumption, the output need not be a full matrix, but can be a "lattice" of qarrays, provided that there are convenient mechanisms for quickly recalculating the missing interior values of the output matrix/lattice, should they be requested by the user. The lattice is composed of "lathes": "slats" (horizontal pieces) and "struts" (vertical pieces) that frame open spaces. This lattice approach provides several key benefits. First, it is more resource-efficient than an alternative full-matrix design, because memory used in calculating each successive lathe can

```
typedef void (*wave_f) (const void *restrict left,
                        const void *restrict leftdown,
                        const void *restrict down,
                        void *restrict out);

typedef struct qt_wavefront_lattice_s qt_wavefront_lattice;

qt_wavefront_lattice *qt_wavefront(qarray * restrict const left,
                                   qarray * restrict const below,
                                   wave_f func);
void qt_wavefront_print_lattice(const qt_wavefront_lattice *const L);
void qt_wavefront_destroy_lattice(qt_wavefront_lattice *const L);
void *qt_wavefront_query_lattice(const qt_wavefront_lattice *const L,
                                 size_t x, size_t y);
```

Figure 6.13. The Qthread Library's Wavefront Interface

be re-used for other lathes. Because of that efficiency, larger problems can be calculated than would otherwise fit into memory. Secondly, each lathe can be a qarray, thereby giving each slat and strut a location. This design is illustrated in Figure 6.14. The struts of the lattice are dark gray, and the slats are light gray; each individual strut or slat is drawn with a thick black border. Vertical sets of struts and horizontal sets of slats are both stored as qarrays, where the size of each strut or slat is defined by the way the qarray segments itself into location-contiguous sections. This lattice defines the size of work units as well: each pair of left-hand strut and lower slat is used to generate the next strut to the right and the next higher slat.

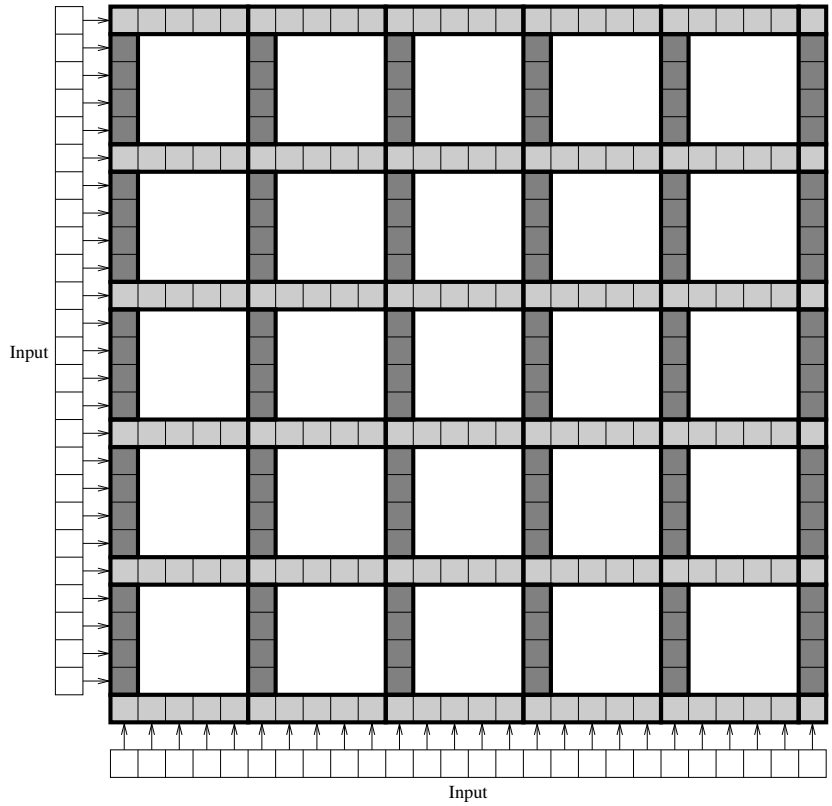


Figure 6.14. The Wavefront Lattice Design

6.4.2 Performance

Each lathe's computation executes in a location that is informed by the location of its inputs: the left input strut and the lower input slat. The location of the output is defined by the location the computation that generated it.

One of the interesting aspects of the Wavefront abstraction is that while its design is inherently parallel, that parallelism is inherently limited. The abstraction gets its name from the way that computation progresses across the output matrix: a leading diagonal edge of computable units whose inputs are ready. As such, the maximum parallelism is the number of elements in that

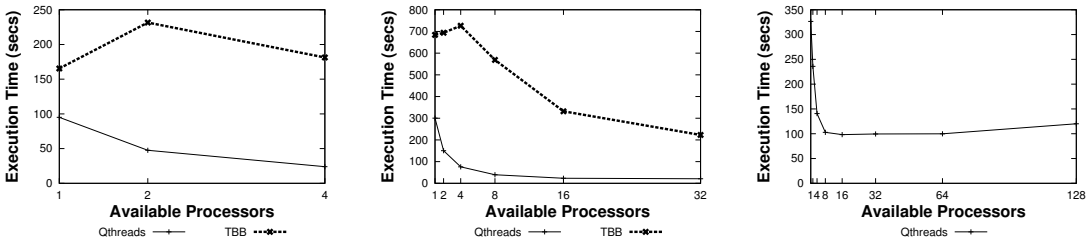
leading edge, which is equal to the number of elements in the smaller of the two input sets — the vertical edge or one less than the horizontal input. This can be seen in Figure 6.12, where the maximum logical parallelism for a five-by-five output matrix is five.

6.4.2.1 Benchmark

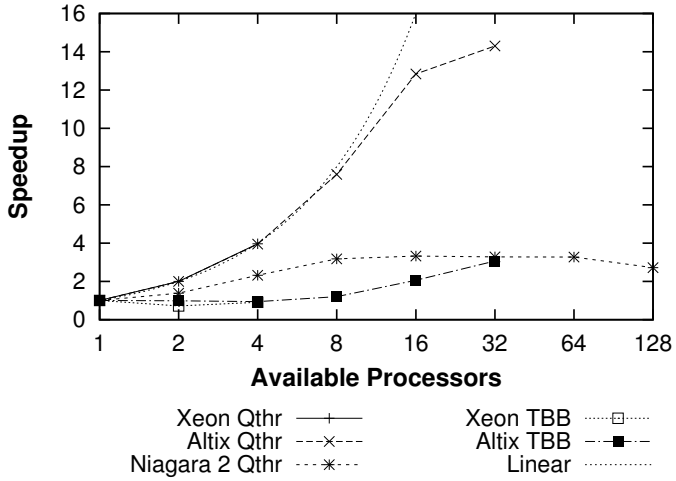
A benchmark was designed to illustrate the performance characteristics of the Wavefront framework. Two 70,000-element qarrays of floating point numbers are initialized randomly and used as input to the `qt_wavefront()` function, along with a function to compute the average of the three inputs (left, below, and diagonally left and below). The `qt_wavefront()` function creates a lattice of qarrays when run, and returns it to the caller. With 70,000-element input sets, this lattice represents approximately 36 GB of data. For comparison, a similar benchmark using a naïve approach similar to the one in Figure 6.12 was implemented with Intel's Threading Building Blocks.

6.4.2.2 Results

Both benchmarks were run on the three machines discussed in the previous chapter, ten times each. Figure 6.15 presents the average execution times from both benchmarks. Clearly, the qthread-based implementation is more efficient than the naïve approach. Besides the lack of scaling of the TBB-based implementation, what is perhaps most noticeable about these graphs is the dramatically different performance on the Niagara 2, which never scales above approximately four times single-threaded speed. Indeed, there is a point at which more parallelism produces less beneficial results.



(a) Xeon (b) Altix (c) Niagara 2



(d) Scaling

Figure 6.15. Wavefront, Generating a $70,000 \times 70,000$ Lattice

The poor scaling on the Niagara 2 is likely a result of a combination of the fact that the Niagara 2 has only one FPU per core, while each core can handle eight threads, and insufficient work. In a $70,000 \times 70,000$ matrix, the work is divided by the qarray into single-page localized segments of between 1024 (on the Niagara 2) and 2048 (on the Altix) elements, limiting the exploitable parallelism to somewhere between 34 and 68 units at peak. Because the wavefront is iterating across a square matrix, most of the time there are less than the peak number of units available to execute at any given time. Assuming uniform ex-

ecution times, over the course of calculating a 68-unit-by-68-unit lattice, as is used on the Niagara 2, there are an average of 34.3 work units available. Thus, scaling much beyond 32 processors is difficult without breaking up those work units. On the Altix, which uses a 34-by-34 lattice, an average of 17.3 work units are available, which explains the falloff in scaling after 16 processors. Complicating matters, when there are a large number of unoccupied worker threads, work is more likely to be consumed by a distant non-local worker looking for work—i.e. the work gets "stolen"—even if there are closer consumers that will be ready soon, which reduces performance further. Additional parallel work units could be made available by making the output lattice more fine-grained, but doing so increases the state requirements, and limits the problem size that can be computed. Halving the size of qarray segments doubles the amount of memory required to store the lathes of the lattice. One possibility for future research is to use a multi-scale approach, allowing for relatively fine-grained computation of each component of the lattice.

6.5 Conclusion

The computational templates presented in this chapter bring together all of the concepts and technologies described in this dissertation. The templates are designed for use in a large-scale multithreaded environment such as was discussed in Chapter 1. They adapt to a variety of memory topologies by using the qthread lightweight threading API with lightweight synchronization and explicitly integrated locality, introduced in Chapter 3. They provide the programmer with computational and communication patterns that have an explicit structure, as discussed in Chapter 4, that must be adapted to the under-

lying system's characteristics. The last two templates adapt their structure to the topology of the system by relying on distributed data structures presented in Chapter 5—both to allocate memory in specific locations and to distribute input-defined work units to location-specific worker threads.

The computational frameworks in this chapter present an archetype of programming future-proof parallel applications so that they can adapt to new memory topologies, architectures, and synchronization mechanisms, exposing the full measure of algorithmic parallelism to be exploited, while being easy to understand, program, and debug.

CHAPTER 7

CONCLUSION

7.1 Recapitulation

The future of computing is undoubtedly parallel. Despite the best efforts of hardware designers, the fundamental bottleneck of the von Neumann computational model has only become more troublesome over time and efforts to redesign shared-memory computers to address the bottleneck typically rely on fine-grained parallelism. Hardware-supported fine-grained parallelism suggests a union between scheduling and layout that will likely provide a moving target to applications, with every execution having a different topology. These developments reveal the semantic disconnect between hardware-supported parallelism features and software-accessible parallelism features.

This thesis advocates a new fine-grained threading approach that provides access to hardware-supported parallelism features. These features—including locality specification, topology querying, and atomic operations—cannot be perfectly used by the compiler and must be both explicitly available to the programmer and understood by the scheduler. A fine-grained threading interface that integrates hardware features enables accurate mapping and adaptation of application algorithmic structure to hardware system topology.

Chapter 3 presented a practical fine-grained threading API and implementation, qthreads. The qthread design provides a coherent interface to the wide variety of lightweight threading architectures and operating system topology and CPU affinity interfaces. The library-based implementation trades some semantic convenience for quick integration with existing applications and a smaller impact on the toolchain. The benefits of this design were illustrated with modified versions of the HPCCG and MTGL-based benchmarks.

Once a fine-grained threading interface is available, creating applications that adapt to hardware topology requires understanding the structure of the application or algorithm being used. Chapter 4 presented the ThreadScope visualization tool to assist in understanding this parallel structure — a structure that involves all forms of communication, including not only explicit synchronization but thread creation and destruction. This type of analysis also provides a new way of approaching basic parallel correctness issues such as bottlenecks, deadlocks, and race conditions.

The impact of application structure on portable performance can be demonstrated most directly with distributed data structures. Chapter 5 presented several basic distributed data structure designs that leverage the qthread threading model to use memory and system resources effectively. The three distributed data structures — a pool, an array, and a queue — achieve scalable performance in a variety of different parallel environments with a variety of different topologies.

Finally, Chapter 6 presented several large scale application structures used in a variety of different fields that use the structural understanding encoded into the distributed data structures to adapt their overall behavior to the hard-

ware topology. The computational templates presented encode parallel computation in a way that allows the implementation to adapt the execution of the computation to the topology and resources available.

7.2 Future Work

As hardware support for lightweight threading interfaces continues to develop, there will continue to be opportunities for expanding the understanding of threaded applications.

7.2.1 Programming Interface

The qthread idea — API and implementation — is to provide a low-level means of using hardware features and obtaining information about the computer in a way that is portable across multiple hardware architectures, topologies, and operating systems. It is not, however, clear that any library-based threading interface is the most convenient interface for expressing parallel intent. While a library-based interface allows applications to be parallelized piece-by-piece, the semantic limitations of a library-based API can make adapting old applications to use parallel loops somewhat awkward. For example, packaging persistent data for threads to access must be explicit rather than implicit. As such, there is work to be done in adapting other threading interfaces, such as OpenMP or Cilk, to use qthreads as their threading back-end. This would provide a certain amount of freedom to the programmer as well as a way of extending those interfaces to include locality information while improving their scheduling mechanism.

The picture of the hardware topology provided to the programmer by existing topology libraries, including the qthread API, is somewhat simplistic. Accounting for things like asymmetric connections and multiple layers of memory within a single node are difficult, if not impossible. It is not clear what a good design would be that could encapsulate the full range of potential structural situations. Even basic information, like cache line size, is difficult to determine reliably or portably. An improved, portable method for determining this information is desperately needed, but presents a significant design and implementation challenge.

7.2.2 Debugging and Tuning

The ThreadScope tool has begun the task of using structural analysis to locate and identify potential threading problems. Problems that can be identified already are bottlenecks, race conditions, and deadlocks. This, of course, is only a small subset of the parallel programming issues that could be identified. Different synchronization mechanisms often have their own unique potential problems that can be identified structurally, and each threaded operation has a different cost depending on the threading model.

Cost estimates can be used to locate situations where the threaded program may need to alter its behavior to take best advantage of the given hardware. For example, though a given loop may be fully parallelizeable, creating a separate thread for each iteration may be unwise if each iteration is sufficiently short. If the average time to create and destroy a thread is x, and the time to execute a single loop iteration is y, a threaded loop of n iterations will only achieve improved performance if $n \times y < (n \times x) + y$, assuming that n threads

can run in parallel. Additionally, if each thread unit can be given a weight—perhaps represented by relative size—load imbalances can be caught.

Additional information about the system on which the application will run can be used to identify potential problem as well. For example, each memory object may be shared by multiple threads. On a shared-cache multicore system, this can be performed efficiently, but there are limits. If more threads than cores with shared cache must access a given memory object, there is the possibility that there will be cache coherency overhead as the object is transferred from cache to cache, resulting in a system-specific bottleneck. With topology information, suggestions can be made about the mapping of application structure to machine structure.

One of the synchronization approaches that is not addressed in this work is transactional memory. Transactions have an impact on application structure similar to the impact of word-sized atomic operations, but can involve a large number of memory objects. Resolving conflicts between overlapping large atomic operations is a complex process, and a good visual representation of that process would greatly assist in designing efficient transactional applications.

7.2.3 Data Structures

There are an infinite number of different data structures that can be redesigned for a distributed setting, from hash tables and binary lookup trees to heaps to unstructured graphs. Work designing such structures will never be done. Even simple extensions to the data structures presented here are worthy of further study.

For example, there are a variety of array operations typical of scientific computing that benefit from locality-awareness but that have not been examined in this work. Array stencils are common in signal processing, image processing, and solving partial differential equations. Applying a naïve stencil algorithm to a distributed array is relatively straightforward, but with foreknowledge of the stencils that will be used, the layout of the array could be improved to align stencil boundaries with segment boundaries and spawning threads for stenciled-out array ranges could be avoided.

Another common array operation is the combination of arrays, such as in string comparison, genetic research, or matrix multiplication — especially with multi-dimensional arrays. These array combination operations have memory access patterns that are well-understood, and provide opportunities for locality-aware optimization. For example, in matrix multiplication, assuming fixed memory affinity, the process can be broken into two sets of operations: those where the components of both arrays are local and those where one of the components is more distant. To optimize bus contention, performing the operations that involve non-local memory access must be balanced with operations that only require local memory access, and these operations can be ordered to optimize cache use.

7.2.4 Computational Templates

There are also a great number of available computational templates that can be used to assist programmers in creating efficient software, and new templates are being designed all the time. For example, MapReduce [43] is a powerful data-processing abstraction recently popularized by Google. The power

of this abstraction is its inherent asynchronicity and easily pipelined design. MapReduce is commonly implemented with a "master" node that assigns tasks to other nodes. Another way of implementing MapReduce is with distributed queues for transporting data between each stage of the MapReduce pipeline. A distributed queue can encourage the preservation of locality between the pipeline stages without a master server. This design may be useful for exploring the best number of workers at each pipeline stage, the optimal method for transferring information between workers, and the effect of worker loss and worker migration.

7.3 Postscript

> When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.
> —*Edsger Dijkstra [47]*

The challenge presented to each new generation of computer scientists is to fully exploit the computers of yesterday while designing the computers of tomorrow. There will probably always be a future-portability problem, but designing applications and algorithms that adapt to the characteristics of the systems they use is a major step toward mitigating the gigantic programming problem posed by each new and more powerful machine.

BIBLIOGRAPHY

1. T. S. Abdelrahman and T. N. Wong. Compiler support for array distribution on NUMA shared memory multiprocessors. *Journal of Supercomputing*, 12(4):349–371, 1998. ISSN 0920-8542. doi:10.1023/A:1008035807599.

2. I. Aelion. cprops - C prototyping tools. http://cprops.sourceforge.net, March 2009.

3. A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, B. H. H. Lim, K. M. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *ISCA '98: 25 years of the international symposia on Computer Architecture (selected papers)*, pages 509–520, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-058-9. doi:10.1145/285930.286009.

4. S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19 (8):26–34, 1986. ISSN 0018-9162. doi:10.1109/MC.1986.1663305.

5. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0β edition, March 2007.

6. R. Altherr, R. Du Bois, L. Hammond, and E. Miller. Software performance tuning with the Apple CHUD tools. *IEEE International Symposium on Workload Characterization*, 0:1, 2006. doi:10.1109/IISWC.2006.302722.

7. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. J. Smith. The Tera system. *Tera Computer Company*, 1999.

8. C. S. Ananian, K. Asanovi , B. C. Kuzmaul, C. E. Leiserson, and S. Lie. Unbounded transational memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 319–327, Washington, DC, USA, February 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi:10.1109/HPCA.2005.41.

9. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support fot the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992. ISSN 0734-2071. doi:10.1145/146941.146944.

10. J. K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. PhD thesis, University of Washington, Seattle, WA, USA, 1987.

11. R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster i/o with river: making the fast case common. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22, New York, NY, USA, 1999. ACM. ISBN 1-58113-123-2. doi:10.1145/301816.301823.

12. J. L. Baer. 2k papers on caches by y2k: Do we need more? Keynote address at the 6th International Symposium on High-Performance Computer Architecture, January 2000.

13. M. Bedy, S. Carr, X. Huang, and C.-K. Shene. A visualization system for multithreaded programming. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 1–5, New York, NY, USA, 2000. ACM. ISBN 1-58113-213-1. doi:10.1145/330908.331798.

14. J. W. Berry, B. A. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the International Parallel & Distributed Processing Symposium*. IEEE, 2007.

15. J. Berthold and R. Loogen. Visualizing parallel functional program runs: Case studies with the eden trace viewer. In *Parallel Computing: Architectures, Algorithms and Applications*, pages 121–128. John von Neumann Institute for Computing, 2007.

16. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 1995. PPOPP '95, ACM Press. ISBN 0-89791-701-6. doi:10.1145/209936.209958.

17. H. J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6. doi:10.1145/1065010.1065042.

18. J. B. Brockman, P. M. Kogge, S. Thoziyoor, and E. Kang. PIM lite: On the road towrds relentless multi-threading in massively parallel systems. Technical Report TR-03-01, Computer Science and Engineering Department, University of Notre Dame, 384 Fitzpatrick Hall, Notre Dame IN 46545, February 2003.

19. J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A low cost, multithreaded processing-in-memory system. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 16–22, New York, NY, USA, 2004. ACM Press. ISBN 1-59593-040-X. doi:10.1145/1054943.1054946.

20. P. Budnik and D. J. Kuck. The organization and use of parallel memories. *IEEE Transactions on Computers*, C-20:1566–1569, December 1971.

21. A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Technical report, Institute for Advanced Study, Princeton University, New Jersey, June 1946. URL `http://hdl.handle.net/2027.42/3972`.

22. C. Ca caval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008. ISSN 1542-7730. doi:10.1145/1454456.1454466.

23. D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60. Institute of Electrical and Electronics Engineers, April 2004. ISBN 0-7695-2151-7. doi:10.1109/HIPS.2004.1299190.

24. B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2004. ATEC '04, USENIX Association.

25. D. F. Carr. How Google works. *Baseline*, July 2006. URL `http://www.baselinemag.com/c/a/Infrastructure/How-Google-Works-1/`.

26. L. Ceze, J. Tuck, J. Torrellas, and C. Ca caval. Bulk disambiguation of speculative threads in multiprocessors. In *ISA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2608-X. doi:10.1109/ISCA.2006.13.

27. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004.

28. J. Chapin, A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 1–13, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-695-6. doi:10.1145/223587.223588.

29. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcio lu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi:10.1145/1094811.1094852.

30. S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, 2009. ISSN 0272-1732. doi:10.1109/MM.2009.34.

31. E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971. ISSN 0360-0300. doi:10.1145/356586.356588.

32. R. Cole. Parallel merge sort. In *SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 511–516, Washington, DC, USA, 1985. IEEE Computer Society Press. ISBN 0-8186-0740-8. doi:10.1109/SFCS.1986.41.

33. M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):369–408, 1963. ISSN 0001-0782. doi:10.1145/366663.366704.

34. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, second edition, 2001.

35. I. Corporation. Intel c++ stm compiler, prototype edition 3.0. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/, August 2009.

36. I. Corporation. Intel thread checker. http://www.intel.com/support/performancetools/threadchecker/, April 2009.

37. D. W. Craig. *Nanothreads: flexible thread scheduling*. Dissertation, University of Illinois at Urbana-Chamaign, Champaign, IL, USA, 2002.

38. Cray. Cray MTA-2 system - HPC technology initiatives, November 2006. URL `http://www.cray.com/products/programs/mta_2/`.

39. Cray. Cray XMT platforrm. http://www.cray.com/products/xmt/index.html, October 2007.

40. D. C. Cronk. *Dynamic Load Balancing via Thread Migration*. PhD thesis, The College of William and Mary, 1999.

41. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. ISSN 0164-0925. doi:10.1145/115372.115320.

42. P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *ACM SIGPLAN Notices*, 41(11):336–346, 2006. ISSN 0362-1340. doi:10.1145/1168918.1168900.

43. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

44. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005. ISSN 1058-9244.

45. P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970. ISSN 0360-0300. doi:10.1145/356571.356573.

46. D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

47. E. W. Dijkstra. Visuals for BP's Venture Research Conference. URL `http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD963.PDF`. circulated privately, June 1986.

48. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *ACM SIGPLAN Notices*, 34(5):229–241, 1999. ISSN 0362-1340. doi:10.1145/301631.301670.

49. J. E. Dorband. Sort computation. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 137–141. IEEE Computer Society Press, October 1988. ISBN 0-8186-5892-4. doi:10.1109/FMPC.1988.47442.

50. U. Drepper and I. Molnar. The native POSIX thread library for linux. Technical report, RedHat Inc., February 2005.

51. T. H. Dunigan. *Denelcor HEP Multiprocessor Simulator*, June 1986.

52. A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 29–42, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-343-3. doi:10.1145/1182807.1182811.

53. S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 1, pages 377–381, August 1991.

54. S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 257–270, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-300-0. doi:10.1145/70082.68206.

55. T. El-Ghazawi and L. Smith. UPC: Unified parallel C. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi:10.1145/1188455.1188483.

56. R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

57. P. Erd s and A. Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

58. D. Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, 1974. ISSN 0001-0782. doi:10.1145/361179.361195.

59. R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, 1986. ISSN 0734-2071. doi:10.1145/214419.214422.

60. S. Fortune and J. Wyllie. Parallelism in random access machines. *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978. doi:10.1145/800133.804339.

61. M. P. I. Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications (Special Issue on MPI)*, 8 (3/4), 1994.

62. E. E. Frank and R. A. Aydt. *The PABLO Performance Visualization System Functional Specification*. Department of Computer Science, University of Illinois, February 1995.

63. F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 88, Washington, DC, USA, 1998. PACT '98, IEEE Computer Society Press. ISBN 0-8186-8591-3.

64. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software—Practice & Experience*, 30(11):1203–1233, 2000. ISSN 0038-0644. doi:10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.3.CO;2-E.

65. W. Gloger. Dynamic memory allocator implementations in Linux system libraries. http://www.dent.med.uni-muenchen.de/ wmglo/, May 1997. URL `http://www.dent.med.uni-muenchen.de/~wmglo/`.

66. S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996. URL `citeseer.ist.psu.edu/article/goldstein96lazy.html`.

67. D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

68. B. Gu, Y. Kim, J. Heo, and Y. Cho. Shared-stack cooperative threads. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied Computing*, pages 1181–1186, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-480-4. doi:10.1145/1244002.1244258.

69. M. Hall, P. M. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. B. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-091-0. doi:10.1145/331532.331589.

70. D. T. Harper, III and J. R. Jump. Vector access performance in parallel memories using skewed storage scheme. *IEEE Transactions on Computers*, 36(12):1440–1449, 1987. ISSN 0018-9340.

71. K. Harzallah and K. C. Sevcik. Predicting application behavior in large scale shared-memory multiprocessors. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 53, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-816-9. doi:10.1145/224170.224356.

72. C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 293–298, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-142-3. doi:10.1145/800055.802046.

73. M. T. Heath and J. E. Finger. ParaGraph: A tool for visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, 1994.

74. M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3):558–588, May 1988. ISSN 0196-5204. doi:10.1137/0909037.

75. J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 1996. ISBN 1-55860-329-8.

76. M. Heroux. Mantevo. http://software.sandia.gov/mantevo/index.html, December 2007. URL `http://software.sandia.gov/mantevo/index.html`.

77. M. Heroux, R. Bartlett, V. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, et al. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

78. W. D. Hillis and L. W. Tucker. The CM-5 connection machine: a scalable supercomputer. *Communications of the ACM*, 36(11):31–40, 1993. ISSN 0001-0782. doi:10.1145/163359.163361.

79. M. Holliday and M. Stumm. Performance evaluation of hierarchical ring-based shared memory multiprocessors. *IEEE Transactions on Computers*, 43(1):52–67, 1994. ISSN 0018-9340. doi:10.1109/12.250609.

80. C. Holt, J. P. Singh, and J. L. Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 134–145, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-786-3. doi:10.1145/232973.232988.

81. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2004. ISBN 0321228626 9780321228628.

82. R. A. Iannucci, G. R. Gao, R. H. Halstead, Jr., and B. J. Smith. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, August 1994.

83. *IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1)*. Institute of Electrical and Electronics Engineers, 1990.

84. *Intel® Threading Building Blocks*. Intel Corporation, 1.10 edition, 2008. URL `http://softwarecommunity.intel.com/isn/downloads/softwareproducts/pdfs/301114.pdf`.

85. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-636-3. doi:10.1145/1272996.1273005.

86. D. Iseminger. *Microsoft WIN32 Developer's Reference Library*. Microsoft Press, Redmond, WA, USA, 1999. ISBN 0735608164.

87. A. Jannesari and W. F. Tichy. On-the-fly race detection in multi-threaded programs. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–10, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-052-4. doi:10.1145/1390841.1390847.

88. S. Jenks and J.-L. Gaudiot. An evaluation of thread migration for exploiting distributed array locality. In *HPCS '02: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and*

Applications, page 190, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1626-2.

89. M. Jeon and D. Kim. Parallel merge sort with load balancing. *International Journal of Parallel Programming*, 31(1):21–33, February 2003. ISSN 0885-7458. doi:10.1023/A:1021734202931.

90. D. Jiang, B. O'Kelley, X. Yu, S. Kumar, A. Bilas, and J. P. Singh. Application scaling under shared virtual memory on a cluster of SMPs. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 165–174, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-164-X. doi:10.1145/305138.305190.

91. T. Johnson. Designing a distributed queue. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 304–311, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7195-5.

92. T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit power efficient Sparc soc (Niagara2). In *ISPD '07: Proceedings of the 2007 International Symposium on Physical Design*, pages 2–2, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-613-4. doi:10.1145/1231996.1232000.

93. Y. Kang, M. W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 192–201, Washington, DC, USA, October 1999. IEEE Computer Society. ISBN 0-7695-0406-X. URL `http://citeseer.ist.psu.edu/kang99flexram.html`.

94. J. C. d. Kergommeaux and B. d. O. Stein. Pajé: An extensible environment for visualizing multi-threaded programs executions. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Lecture Notes in Computer Science, pages 133–140, London, UK, September 2000. Springer-Verlag. ISBN 978-3-540-67956-1. doi:10.1007/3-540-44520-X_17.

95. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One level storage system. *IRE Transactions on Electronic Computers*, EC-11 (2):223–235, April 1962.

96. A. Kleen. An NUMA API for Linux. http://halobates.de/numaapi3.pdf, August 2004.

97. K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3-d discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(198), 1992.

98. P. M. Kogge. The EXECUBE approach to massively parallel processing. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages 77–84, Chicago, IL, August 1994.

99. P. M. Kogge, S. Bass, J. B. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

100. P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005. ISSN 0272-1732. doi:10.1109/MM.2005.35.

101. J. S. Kowalik, editor. *On Parallel MIMD computation: HEP supercomputer and its applications*, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology. ISBN 0-262-11101-2.

102. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi:10.1145/359545.359563.

103. L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974. ISSN 0001-0782. doi:10.1145/360827.360844.

104. D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24:1145–1155, December 1975.

105. T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analyzing parallel program executions using multiple views. *J. Parallel Distrib. Comput.*, 9 (2):203–217, 1990. ISSN 0743-7315. doi:10.1016/0743-7315(90)90046-R.

106. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 229–239, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-198-9. doi:10.1145/10590.10610.

107. D. C. Lin, P. W. Dymond, and X. Deng. Parallel merge sort on concurrent-read owner-write pram. In *Euro-Par '97: Proceedings from the Third International Euro-Par Conference on Parallel Processing*, pages 379–383, London, UK, 1997. Springer-Verlag. ISBN 3-540-63440-1.

108. L. B. Linden. Parallel program visualization using ParVis. *Parallel Computer Systems: Performance Instrumentation and Visualization*, pages 157–187, 1990. doi:10.1145/100215.100265.

109. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111. IEEE Computer Society Press, June 1988. ISBN 0-8186-0865-X. doi:10.1109/DCS.1988.12507.

110. C. D. Locke, T. J. Mesler, and D. R. Vogel. Replacing passive tasks with ada9x protected records. *Ada Letters*, XIII(2):91–96, 1993. ISSN 1094-3641. doi:10.1145/152827.152834.

111. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-232-8. doi:10.1145/339647.339673.

112. A. D. Malony and D. A. Reed. *Visualizing parallel computer system performance*, pages 59–90. ACM, New York, NY, 1989. ISBN 0-201-50390-5. doi:10.1145/75705.75709.

113. U. Manber. On maintaining dynamic information in a concurrent environment. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 273–278, New York, NY, USA, 1984. ACM. ISBN 0-89791-133-4. doi:10.1145/800057.808691.

114. X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A library implementation of the nano-threads programming model. In *Euro-Par*, volume 2, pages 644–649, 1996. URL `http://citeseer.ist.psu.edu/martorell96library.html`.

115. S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, page 162, New York, NY, USA, 2004. CF '04, ACM Press. ISBN 1-58113-741-9. doi:10.1145/977091.977115.

116. A. Meyer and J. G. Riecke. Continuations may be unreasonable. In *LFP '88: Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 63–71, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-273-X. doi:10.1145/62678.62685.

117. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of*

the fifteenth annual ACM symposium on Principles of distributed computing, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. doi:10.1145/248052.248106.

118. S. W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, June 1996.

119. M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. *ACM SIGPLAN Notices*, 41(11):359–370, 2006. ISSN 0362-1340. doi:10.1145/1168918.1168902.

120. C. Moretti, J. Bulosan, D. Thain, and P. Flynn. All-pairs: An abstraction for data intensive cloud computing. In *IPDPS '08: Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, April 2008. ISBN 978-1-4244-1693-6. doi:10.1109/IPDPS.2008.4536311.

121. J. P. Morrison. Data responsive modular, interleaved task programming system. Technical Disclosure Bulletin 8, IBM, January 1971.

122. R. C. Murphy. *Travelling Threads: A New Multithreaded Execution Model*. PhD thesis, University of Notre Dame, Notre Dame, IN, USA, June 2006.

123. R. C. Murphy, P. M. Kogge, and A. F. Rodrigues. The characterization of data intensive memory workloads on distributed PIM systems. *Lecture Notes in Computer Science*, 2107:85–??, 2001. URL `http://citeseer.ist.psu.edu/murphy00characterization.html`.

124. W. E. Nagel, A. Arnold, M. Weber, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

125. C. Natarajan, S. Sharma, and R. K. Iyer. Measurement-based characterization of global memory and network contention, operating system and parallelization overheads. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5510-0. doi:10.1145/191995.192018.

126. NeoMagic. Neomagic products, 2001. URL `http://www.neomagic.com`.

127. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi:10.1145/1250734.1250746.

128. J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546, London, UK, 1999. Springer-Verlag. ISBN 3-540-65831-9.

129. J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and J. Ju. *The Global Arrays User's Manual*, 2006.

130. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

131. J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The armci approach. *International Journal of High Performance Computing Applications*, 20(2): 233–253, 2006. ISSN 1094-3420. doi:10.1177/1094342006064504.

132. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and D. Ayguadé. Is data distribution necessary in openmp? In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 47, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

133. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Leveraging transparent data distribution in OpenMP via user-level dynamic page migration. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 415–427, London, UK, 2000. Springer-Verlag. ISBN 3-540-41128-3.

134. R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. ISSN 1061-7264. doi:10.1145/289918.289920.

135. Y. Nunomura, T. Shimizu, and O. Tomisawa. M32R/D-integrating dram and microprocessor. *IEEE Micro*, 17(6):40–48, 1997. ISSN 0272-1732. doi:10.1109/40.641595.

136. *The Open Group Technical Standard Base Specifications, Issue 7 (POSIX.1-2008)*. The Open Group, January 2008.

137. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, 2.5 edition, May 2008.

138. M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8491-7. doi:10.1145/279358.279387.

139. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

140. D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-281-X.

141. D. A. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997. ISSN 0272-1732. doi:10.1109/40.592312.

142. J. Perdue. Predicting performance on SMPs. a case study: The SGI power challenge. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 729, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0574-0.

143. E. Pietriga. Zgrviewer, a graphviz/dot viewer. http://zvtm.sourceforge.net/zgrviewer.html, April 2009.

144. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, chapter 2, pages 63–82. Cambridge University Press, Cambridge, England, 2nd edition, September 1992.

145. Python. Stackless python. http://www.stackless.org, January 2008.

146. J. Qin, K. Y. Chan, and P. Manneback. Performance analysis in parallel triangular solver. In *ICAPP '96: Proceedings of the Second International Conference on Algorithms and Architectures for Parallel Processing*, pages 405–412, June 1996. doi:10.1109/ICAPP.1996.562902.

147. S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3.

148. A. F. Rodrigues, R. C. Murphy, P. M. Kogge, and K. D. Underwood. The structural simulation toolkit: Exploring novel architectures. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 157, New York, NY, USA, 2006. SC '06, ACM Press. ISBN 0-7695-2700-0. doi:10.1145.1188455.1188618.

149. S. Rosen. Electronic computers: A historical survey. *ACM Computing Surveys*, 1(1):7–36, 1969. ISSN 0360-0300. doi:10.1145/356540.356543.

150. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3): 217–298, 2002. ISSN 0164-0925. doi:10.1145/514188.514190.

151. S. Saini and H. D. Simon. Applications performance under OSF/1 AD and SUNMOS on Intel Paragon XP/S-15. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 580–589, New York, NY, USA, 1994. ACM Press. ISBN 0-8186-6605-6. doi:10.1145/602770.602868.

152. A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *International Symposium on Computer Architecture*, May 1996.

153. S. E. Sevcik. An analysis of uses of coroutines. Dissertation, The University of North Carolina at Chapel Hill, 1976.

154. N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-710-3. doi:10.1145/224964.224987.

155. Y. Shiloach and U. Vishkin. An o(n log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

156. J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Boston, MA, USA, 2002. ISBN 0-201-72914-8.

157. A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 6 edition, 2003. ISBN 0-471-25060-0.

158. J. P. Singh, E. Rothberg, and A. Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 189–199, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-671-9. doi:10.1145/181014.181329.

159. A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 171–180, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-659-X. doi:10.1145/183018.183038.

160. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. ISSN 1058-9244.

161. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

162. B. So, M. W. Hall, and H. E. Ziegler. Custom data layout for memory parallelism. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 291, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

163. I. Software. Rational purify. http://www.ibm.com/software/awdtools/purify/, April 2009.

164. J. T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Georgia Institute of Technology, Atlanta, GA, January 1995.

165. Q. F. Stout, D. L. De Zeeuw, T. I. Gombosi, C. P. T. Groth, H. G. Marshall, and K. G. Powell. Adaptive blocks: A high performance data structure. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–10, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-985-8. doi:10.1145/509593.509650.

166. W. T. Sullivan, III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on project SERENDIP data and 100,000 personal computers. In C. Batalli Cosmovici, S. Bowyer, and D. Werthimer, editors, *IAU Colloquim 161: Astronomical and Biochemical Origins and the Search for Life in the Universe*, page 729, Bologna, Italy, January 1997. Editrice Compositori.

167. *Memory and Thread Placement Optimization Developer's Guide*. Sun Microsystems, Inc., Santa Clara, CA, June 2007. URL `http://dlc.sun.com/osol/docs/content/MTPODG/docinfo.html`.

168. F. Szelényi and W. E. Nagel. A comparison of parallel processing on Cray X-MP and IBM 3090 VF multiprocessors. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 271–282, New York, NY, USA, 1989. ACM. ISBN 0-89791-309-4. doi:10.1145/318789.318819.

169. J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs, 2001. URL citeseer.ist.psu.edu/tao01memory.html.

170. J. Tao, W. Karl, and M. Schulz. Using simulation to understand the data layout of programs, 2001. URL citeseer.ist.psu.edu/tao01using.html.

171. O. M. Team. Portable linux processor affinity. http://www.open-mpi.org/projects/plpa/, March 2009.

172. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 578–587, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-2630-5.

173. O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing*, pages 410–419, 1993. URL http://citeseer.ist.psu.edu/temam93to.html.

174. O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 261–271, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-659-X. doi:10.1145/183018.183047.

175. D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and A. J. G. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 11, pages 299–335. John Wiley & Sons, Inc., 2003.

176. M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, volume 1, pages 35–41, Los Alamitos, CA, USA, 1988. SC '88, IEEE Computer Society Press. ISBN 0-8186-0882-X. doi:10.1109/SUPERC.1988.44632.

177. R. F. Van der Wijngaart, S. R. Sarukkai, and P. Mehra. Analysis and optimization of software pipeline performance on MIMD parallel computers. *Journal of Parallel and Distributed Computing*, 38(1):37–50, October 1996. ISSN 0743-7315. doi:10.1006/jpdc.1996.0127.

178. V. R. Volkman. Threads for windows. *Windows/DOS Developer's Journal*, 4(9):50–54, 1993. ISSN 1059-2407.

179. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997. ISSN 0018-9162. doi:10.1109/2.612254.

180. K. B. Wheeler and R. C. Murphy. Lightweight threading for architectural design research. In *CSRI Summer Proceedings*, August 2007.

181. K. B. Wheeler and D. Thain. Visualizing massively multithreaded applications with threadscope. *Concurrency and Computation: Practice & Experience*, 2009.

182. K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS '08: Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, pages 1–8. MTAAP '08, IEEE Computer Society Press, April 2008. ISBN 978-1-4244-1693-6. doi:10.1109/IPDPS.2008.4536359.

183. W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. ISSN 0163-5964. doi:10.1145/216585.216588.

184. L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions. In *HPDC '09: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pages 1–10, New York, NY, USA, 2009. ACM Press. ISBN 978-1-60558-587-1. doi:10.1145/1551609.1551613.

185. Q. A. Zhao and J. T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, January 1995.

186. Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *ACM SIGMOD Record*, 34(3):37–43, 2005. ISSN 0163-5808. doi:10.1145/1084805.1084813.

This document was prepared & typeset with LaTeX 2$_\varepsilon$, and formatted with NDdiss2$_\varepsilon$ classfile (v3.0[2005/07/27]) provided by Sameer Vijay.