Title
Optimizing Irregular Data Accesses for Cluster and Multicore Architectures

Author
Su, Jimmy Zhigang

Publication Date
2010

Peer reviewed|Thesis/dissertation

Optimizing Irregular Data Accesses for Cluster and Multicore Architectures

by

Jimmy Zhigang Su

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine A. Yelick, Chair
Professor Rastislav Bodik
Professor Ming Gu

Fall 2010

Optimizing Irregular Data Accesses for Cluster and Multicore Architectures

by

Jimmy Zhigang Su

ABSTRACT

Optimizing Irregular Data Accesses for Cluster and Multicore Architectures

by

Jimmy Zhigang Su

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine A. Yelick, Chair

Applications with irregular accesses to shared state are one of the most challenging computational patterns in parallel computing. Accesses can involve both read or write operations, with writes having the additional complexity of requiring some form of synchronization. Irregular accesses perform poorly in local cached-based memory systems and across networks in global distributed memory settings, because they have poor spatial and temporal locality. Irregular accesses arises in transaction processing, in various system level programs, in computing histograms, performing sparse matrix operations, updating meshes in particle-mesh methods, and building adaptive unstructured meshes. Writing codes with asynchronous parallel updates on clusters and multicore processors presents different sets of challenges. On clusters, the goal is to minimize the number of messages and the volume of messages between nodes. While on multicore machines, the goal is to minimize off-chip accesses since there is significant performance difference between on chip and off chip memory access.

In this dissertation, we explore various analyses, optimizations, and tools for shared accesses on both multicore and distributed memory cluster architectures. On cluster architectures, we consider both irregular reads and writes, demonstrate how Partitioned Global Address Space languages support programming irregular problems, and develop optimizations to minimize communication traffic, both in volume and number of distinct events. On multicore processors, we consider the lower level code generation and tuning problem, independent of any particular source language. We explore performance tradeoffs between various shared update implementations, such as locking, replication of state to avoid collisions, and hybrid versions. We develop an adaptive implementation that adjusts the shared update strategy based on densities that yields significant speedups. In addition, we develop a performance debugging tool to find scalability problems in large scientific applications early in the development cycle. Throughout the thesis we perform experiments demonstrating the value of our optimizations

and tools in both architectural settings, use a set of benchmarks and applications that include histogram making, sparse matrix computations, and two scientific simulations involving particle-mesh methods. Our results show substantial speeds of up to 4.8X for multicore platforms and 120X for clusters. The results are a comprehensive set of techniques for improving the performance of irregular applications using advanced languages, compilers, analyses, optimizations and tools.

_____

Professor Katherine A. Yelick
Dissertation Committee Chair

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Parallel computing has long been the mainstay of high end scientific computing and has recently moved into the mainstream with multicore processors marking the end of single processor performance improvements. Single chips now have up to twelve conventional microprocessor cores, while graphics chips have hundreds of lightweight cores and the largest petascale machines available today have hundreds of thousands of cores. Despite years of effort by the research community, there is no wide-spread acceptance of a single parallel programming model, and the message passing model most popular for high end scientific computing appears unsuitable for the growing number of cores on a single chip. One of the challenges to a single common programming model is the wide spectrum of parallel programming patterns and the trade-off between generality needed to support all of them and the simplicity that is possible for a narrower domain. Data parallel languages allow programmers to write expressions that operate on aggregate data structures, such as arrays, but preserve much of the intuition behind a serial semantics. But these languages place a heavy burden on the compiler and runtime system, and the elegance of the pure data parallel language quickly breaks down when language designers try to support the breadth of parallel constructs. Message passing languages such as MPI [50] limit the interaction between parallel threads to explicit communication points, making it easier to see and therefore control the interactions. But the explicit, two-sided nature of the communication becomes awkward for some computational problems in which communication events on one thread are unexpected on another. Shared memory models (often with dynamically created threads or OpenMP [53]) offer the generality of being able to build shared data structures and access them conveniently from multiple threads, but they to not scale to large numbers of processors. Partitioned Global Address Space (PGAS) languages provide some of the best features of each of these paradigms in order to make them expressive, convenient, efficient and scalable. We will explore a particularly difficult class of applications—those involving irregular accesses—and describe analyses, optimizations and tools to support these applications.

Applications with irregular accesses to shared state are one of the most challenging computational patterns in parallel computing. The accesses may be irregular in space, accesses locations that are not contiguous, and do not follow a statically predictable pattern. They can also be irregular in time, so they occur asynchronously at times that are not predictable in advanced and are therefore difficult to coordinate if multiple threads need to be involved. Accesses can involved both read or write operations, with writes having the additional complexity of requiring some form of synchronization in case of collisions. Irregular accesses

perform poorly in local cached-based memory systems and across networks in global distributed memory settings.

Irregular accesses arises in transaction processing, in various system level programs, and in algorithms that arise in HPC applications, such as computing histograms, performing sparse matrix operations, updating meshes in particle-mesh methods, and building adaptive unstructured meshes. Writing codes with asynchronous parallel updates on clusters and multicore processors presents different sets of challenges. On clusters, the goal is to minimize the number of messages and the volume of messages between nodes. Similarly, we try to minimize the memory traffic between sockets on multicore machines, since there is significant performance difference between on chip and off chip memory access.

In this dissertation, we explore various analyses, optimizations, and tools for shared updates on both multicore and distributed memory cluster architectures. On cluster architectures, we consider both irregular reads and writes, demonstrate how PGAS languages support programming some problems, and develop optimizations to minimized communication traffic, both in volume and number of distinct events. Our PGAS work is done in a Java-based language called Titanium [37, 74]. We explore optimizations for the irregular data access when it appears as a read, using sparse matrix vector multiply as the benchmark, and as a write, using histogram construction and two particle-mesh methods as benchmarks. On multicore processors, we consider the lower level code generation and tuning problem, independent of any particular source language. We explore performance tradeoffs between various shared update implementations, such as locking, replication of state to avoid collisions, and hybrid versions. The effectiveness of the optimizations is highly dependent on the frequency of updates, the distribution of the accesses across system memory, the likelihood of collisions during updates, and the relative size of the data structures involved. We develop an adaptive implementation that adjusts the shared update strategy based on densities that yields significant speedups. In addition to the difficulty of ensuring correctness on these problems with irregular accesses, there is a problem of selecting the right implementation based on the machine and usage characteristics. The space of reasonable implementation approaches is enormous, when considers all the parameter settings within each algorithm. At various points within the thesis we use performance models and automatic search-based performance tuning to select optimizations. In addition, we develop a performance debugging tool to find scalability problems in large scientific applications earlier in the development cycle.

1.1 Optimizations for Irregular Read Data Access

Irregular data access can appear both as a read or write. We develop new compiler and runtime extensions for the Titanium implementation to support programs with indirect read

array accesses, such as A[B[i]], where the A array may live in a remote processor's memory. The effectiveness of the optimizations is evaluated on three different cluster architectures. These computations arise in sparse iterative solvers, particle-mesh methods, and elsewhere. We add compiler support for an *inspector executor* execution model, which optimizes communication performing runtime optimization based on the dynamic pattern of indices in the indirection array, which is B in the previous example. We explore several possible transformations that can be done on the communication to minimize the number of messages and overlap communication with both computation and other communication. We use the sparse matrix vector multiply benchmark to evaluate the effectiveness of our optimizations.

1.2 Asynchronous Updates Challenges on Cluster

In spite of recent trends towards increased parallelism in all type of computing devices, parallel programming remains a difficult and time-consuming process. The barrier to entry for parallel programming, especially on high end machines, still deters many scientists and engineers from writing parallel programs. Parts of this dissertation were done in the context of Titanium. Titanium is a Partitioned Global Address Space (PGAS) language. Titanium offers a more convenient programming style, by permitting a thread to read and write to shared memory directly. For asynchronous updates, it can be implemented using synchronized region with fine grained accesses inside the region. Without adequate optimization by the compiler, programs written in this fashion would have very poor performance. The straightforward compilation results in four network round trips per update when the memory location is remote: lock acquisition, lock release, remote memory read and write. In practice, Titanium programmers often resort to manual optimization by rewriting their program in a coarse-grained style to aggregate the small messages together. The experience of the Titanium group and others is that performance is paramount in this domain as our experience and others shows that programmers will spend a significant amount of time and code to improve performance, and are unwilling to use languages or tools that results in significant performance penalties. In the first part of the dissertation, we developed analysis and optimizations for programs with asynchronous parallel updates in Titanium, so programmers can benefit from the high level programming language without the performance penalty.

1.3 Asynchronous Update Challenges on Multicore

The vast array of parallel computing devices offers different challenges for optimization. In contrast to cluster, shared memory is accessible by all threads on modern multicore

architectures. This allows the programmer to write parallel programs without using message passing. This is a significant gain in programmability and productivity for the programmer. But the use of shared memory does not alleviate the need for performance tuning entirely. Access from different cores to the shared memory yields different performance due to NUMA issues. This is analogous to the performance difference between local memory access and remote memory access on cluster, although the magnitude of the performance gap is less on multicore processors. The performance model of combining per update cost and overhead cost is applicable to both multicore and cluster. Overhead cost includes time spent in reduction and pre-processing of data points. In this part of the dissertation, we explore various Particle In Cell (PIC) implementations that perform tradeoffs between per update cost and overhead cost.

1.4 Performance Debugging

High performance kernels are necessary building blocks for a large parallel application, but they alone do not make the application usable for the end user on large scale machines. There are parts of the program that are less scrutinized for performance that become obstacles for running experiments on large number of processors or large inputs. As the number of cores increases for future multicore architectures, this problem will also be prevalent in the multicore domain. These include I/O, initialization, and data distribution. During application development and testing on small number of processors and small input sizes, these problems are typically hidden behind long running kernels. When the application is in production, these problems can no longer be ignored as they are not scalable with the number of processors or input size. This causes longer waiting time for the experiments to finish or not finish at all. In this part of the dissertation, we introduce a performance debugging tool for catching these problems earlier in the development cycle in the context of Titanium.

1.5 Thesis Contributions

In this thesis, our primary contributions can be summarized into the following:

1. We develop compiler analysis and optimization for asynchronous parallel updates in the Titanium compiler.

2. We evaluate the synchronization region optimization on the Heart code spread force, histogram, and particle gravitation benchmarks. The generated code using the

optimization achieves speedups of 90x, 120x, and 70x over the sequential code on 128 processors, while the generated code without this optimization does not get speedups above 10x on the same number of processors.

3. We explore the optimization space for PIC code on modern multicore processors. Our results show that optimizations to maximize the number of local updates are necessary to obtain scalable performance. Simple parallel implementations using locking or full replication yield suboptimal performance.

4. We develop an adaptive PIC implementation that maximizes the number of local updates. It achieves a speedup up to 4.8x over locking implementation on modern multicore processors.

5. We developed *inspector-executor* compiler and runtime optimizations for the Titanium implementation to support programs with indirect read array accesses.

6. We analyze the benefits of the automated *inspector-executor* transformation using sparse matrix vector multiplication on a set of matrices from real applications. The speedup relative to MPI on a suite of over 20 matrices averages 21% on three different machines, with the maximum speedup of more than 2x.

7. We developed a performance debugging tool named ti-trend-prof to find scalability problems in parallel code using multiple program runtime traces. ti-trend-prof is able to predict scalability problems at high processor counts or large input sizes using traces from small number of processors or small input sizes.

8. For two of the largest Titanium applications, Heart code [54] and AMR [11], ti-trend-prof found multiple scalability bugs within hours instead of days for manual debugging.

Chapter 2

Background

In this chapter, we present background information on Titanium and GASNet [15]. More details on the Titanium language and GASNet can be found in the Titanium language specification [37] and GASNet specification.

2.1 Titanium

Titanium is a language designed for high-performance parallel scientific computing. Titanium uses Java as its base, thereby leveraging the modern language features of Java such as object oriented programming, strong type checking, memory safety, garbage collection, and a rich library. Titanium has an optimizing compiler with both memory hierarchy optimizations [55] and communication optimizations [62, 63]. The main additions to Java in Titanium are immutable classes, multi-dimensional arrays, an explicitly parallel Single Program Multiple Data (SPMD) model of computation with a global address space, and zone-based memory management [1, 29]. High performance is always the first criterion in any parallel programming language targeting high-performance parallel scientific computing. Titanium's emphasis on programmer productivity as an equally important goal makes it unique from other parallel programming languages in this space. A recent study suggests that Titanium implementations for three of the NAS parallel benchmarks can match the performance of the standard MPI/Fortran implementations, while requiring substantially fewer lines of code [25].

Titanium is a dialect of Java, but does not use the Java Virtual Machine model. Instead, the end target is assembly code. For portability, Titanium is first translated into C and then compiled into an executable. In addition to generating C code to run on each processor, the compiler generates calls to a runtime layer based on GASNet, a lightweight communication layer that exploits hardware support for direct remote reads and writes when possible. Titanium runs on a wide range of platforms including uniprocessors, shared memory machines (SMPs), distributed-memory clusters of uniprocessors or SMPs (CLUMPS), and a number of specific supercomputer architectures (Cray X1, Cray T3E, SGI Altix, IBM SP, Origin 2000, and NEC SX6).

Titanium is a Single Program, Multiple Data (SPMD) language, so all threads execute the same code image. Each thread has its own thread ID, which can be obtained through a call to Ti.thisProc. A call to Ti.barrier causes a thread to wait until all other threads reach the same barrier. Titanium is part of the Partitioned Global Address Space (PGAS) programming language family. The global address space provides a shared memory view of the combined memory of

the processors since that thread running on one processor can directly read or write the memory associated with another. This feature significantly increases programmer productivity, since the programmer can build large shared data structures and does not need to write explicit communication with matching send and receive calls as in message passing programs. The address space is logically partitioned to provide a scalable performance abstraction for the global address space in which some data is considered nearby each processor. The Titanium type system distinguishes between references to data that is nearby (local) vs data that can possibly be remote (global). Figure 2.1 demonstrates the difference between local pointers and global pointers. Local pointers can only point to a memory address that is on the local processor. Global pointers can point to either local memory or remote memory on a remote processor. By default, Titanium pointers are global. Each processor p0 to pn has a section of memory that is viewed as nearby. Data in that memory can further be divided into local and global. Execution of l.x on processor 0 does not require communication, since the object pointed to by l is in the nearby memory. In contrast, execution of g.x on processor 0 would require communication between processor 0 and processor 1, because the object pointed to by g lives on processor 1. A global pointer can point to both local and global objects. On processor 1, the global pointer g points to an object in nearby memory, so the execution of g.x on processor 1 would not require any communication.

Through programmer type annotation and compiler inference [45], some Titanium pointers can be inferred as local at compile time. This can significantly improve serial performance and reduce memory footprint. Global pointers contain a field for node ID in addition to the pointer address.

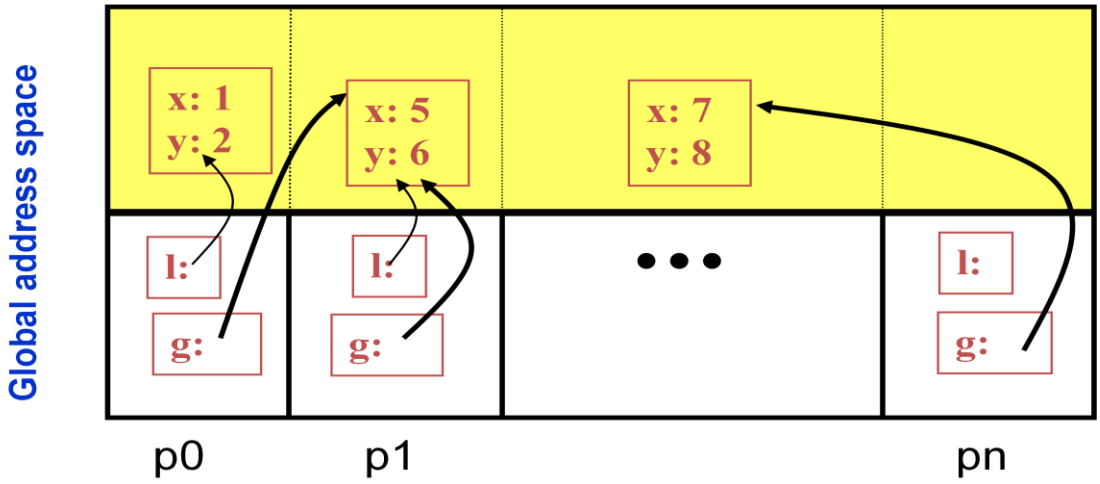


Figure 2.1: Titanium is a Partitioned Global Address Space language. Pointers are separated into local and global pointers. By default, Titanium pointers are global.

Titanium extends sequential Java with the following key features:

1. Titanium arrays – Titanium provides a powerful multidimensional array abstraction defined on a global index space along with the same kinds of sub-array operations available in Fortran 90.

2. Domain calculus – The built-in multidimensional domain calculus provides syntactical support for sub-arrays. In Titanium, the location of an array element is represented by an integer vector called a *point*, and a *domain* is a set of points which can be either rectangular or not. Points (Point) and domains (RectDomain, Domain) are first-class types and literals.

3. Foreach loops – The iteration over any multidimensional Titanium array is expressed concisely in one loop, using the unordered looping construct *foreach*. Unlike for loops, the Titanium compiler has the freedom to execute the loop iterations in any order. The compiler currently does not exploit this as an optimization opportunity.

4. Distributed data structures – Global data structures are built in Titanium through its general pointer-based distribution mechanism. It is each process's responsibility to construct its local share of a global data structure and have the references available to others. After that each process can access the entire data structure in the same way as on a shared-memory machine by dereferencing pointers.

5. Non-blocking array copy – The bulk communication between two processes is realized through a copy method of Titanium arrays. The support for non-blocking array copy enables the overlap of computation and communication. Some of the optimizations presented in this dissertation automatically generate non-blocking array copy calls.

6. The *local* keyword and locality qualification – Titanium expresses the affinity between data and its owning process explicitly using the local type qualifier. The default type for a Titanium pointer is *global*, meaning that the pointer can point to data residing in the local memory or remote memory. Locality information is automatically propagated by the Titanium optimizer using a constraint-based inference. This performance feature helps especially when running computations on distributed-memory platforms. Global pointer contains information on node ID and the actual memory address on that node. Pointer dereferencing using global pointers in a loop often causes the native C compiler to give up on loop optimizations such as loop unrolling, even when the global pointer is pointing to data in local memory.

7. Immutable classes – All Java objects are accessed through references. This adds a constant overhead including extra level of indirection and object creation and destruction to programs with heavy use of small objects. To address this problem in Titanium, application-specific primitive types can be defined as immutable classes. Immutable classes are not extensions of any class including Object. All fields in an immutable class are final. Objects of an immutable class (lightweight objects) are unboxed, analogous to C structs. They are manipulated and passed by value.

2.2 Memory Consistency Model

Optimizations presented in this dissertation require the ability to reorder remote writes in a program through aggregation and non-blocking communication. In a uniprocessor environment, such compiler transformations must adhere to a simple data dependency constraint: the orders of all pairs of conflicting accesses (accesses to the same memory location, with at least one a write) must be preserved. The execution model for parallel programs is considerably more complicated, since each thread executes its own portion of the program asynchronously, and there is no predetermined ordering among accesses issued by different threads to shared memory locations. A memory consistency model defines the memory semantics and restricts the possible execution orders of memory operations. Titanium has a relaxed memory consistency model. Between synchronization events, accesses from one thread may be observed to happen out of order by another thread, and such reordering is allowed under the Titanium memory consistency model. Here are some informal properties of the Titanium model.

· Locally sequentially consistent: All reads and writes issued by a given processor must appear to that processor to occur in exactly the order specified. Thus, dependencies within a processor stream must be observed.

· Globally consistent at synchronization events: At a global synchronization event, such as a barrier, all processors must agree on the values of all the variables. At a non-global synchronization event, such as entry into a critical section, the processor must see all previous updates made using that synchronization event.

The first property implies that a processor must be able to read its own writes. If a processor writes to array elements that have been prefetched from a remote location into a local buffer, subsequent reads by that processor must return the new value. The second property makes data prefetched prior to a synchronization point unusable after that synchronization point. The prefetched data may have been changed by other processors, and reads after the synchronization point must reflect those changes. This property prevents code motion past synchronization points.

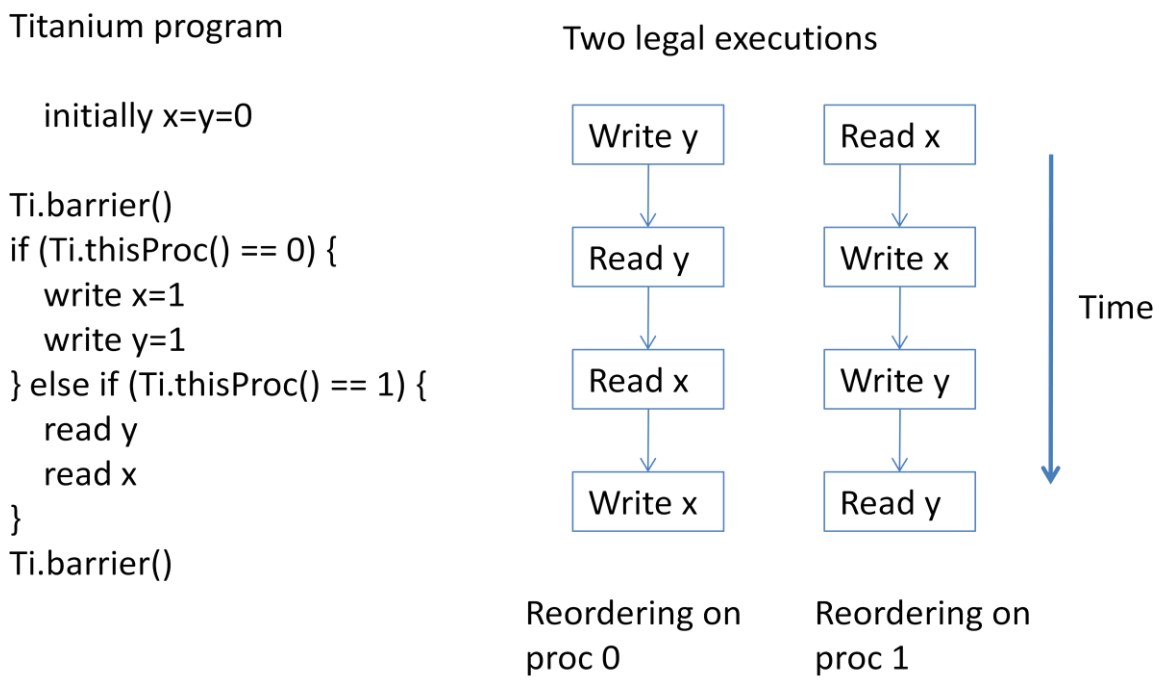


Figure 2.2: Shared memory accesses without intervening synchronization events can be reordered under the Titanium memory consistency model.

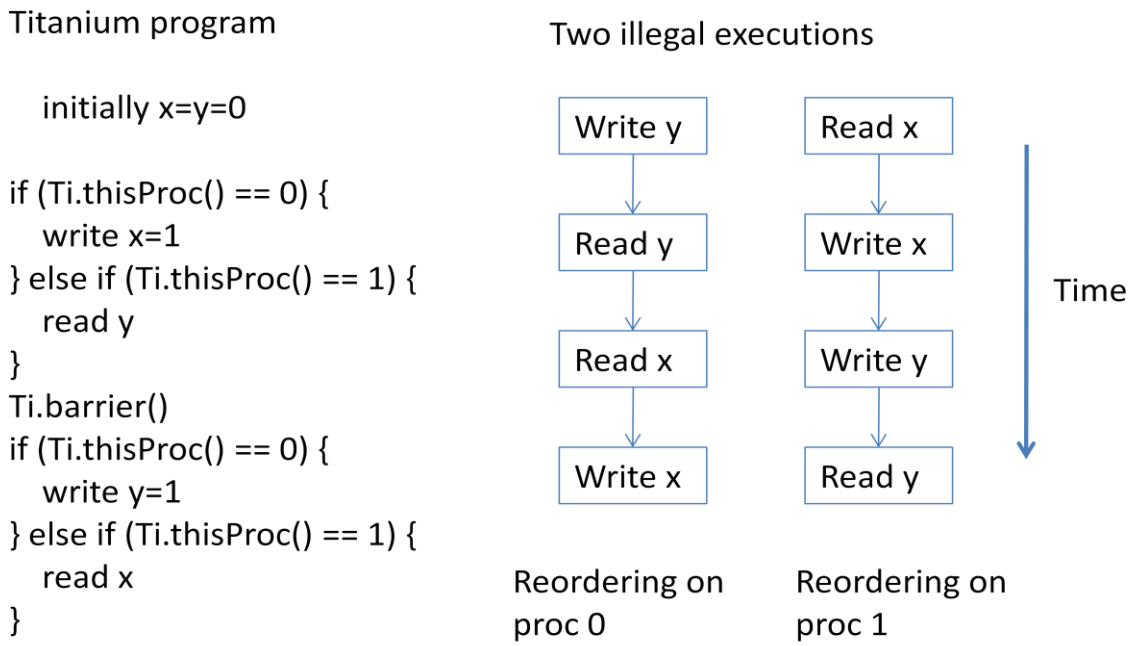


Figure 2.3: Shared memory accesses are not allowed to reorder across barriers under the Titanium memory consistency model.

Figure 2.2 shows a Titanium program where reads and writes can be reordered between a pair of barriers. It is possible for Processor 1 to observe the reordering of writes on Processor 0. The (x,y) pair read by Processor 1 can read any combination of 0's and 1's, including (0,1) as shown by the two reordered executions in Figure 2.2. This is allowed under the Titanium memory consistency model, since there is no synchronization event between those shared memory accesses. In contrast, Figure 2.3 shows a Titanium program where the same reordering of memory accesses across the barrier is not allowed. In this case, the only allowed values read for the (x, y) pair on processor 1 is (1, 0).

2.3 Textually Aligned Barriers

A *barrier* causes a thread to wait until all other threads reach a barrier. Titanium has the unique feature that requires all threads to reach the same textual sequence of barriers during the execution of the program. This property is called textually aligned barriers: a call to a barrier in Titanium causes the calling thread to wait until all other threads have executed the same textual instance of the barrier call. The code in the example below is not allowed because not all the threads will hit the same textual barrier. The Titanium compiler checks statically that all the barriers are lined up correctly.

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  Ti.barrier(); // odd ID threads
```

Figure 2.4: Barriers in Titanium must be textually aligned. Only the even ID threads can hit the first barrier. The Titanium compiler emits a compilation error for this code.

In contrast, if we change the conditional to Ti.numProcs() % 2 == 0, then it would satisfy the textually aligned property, since all threads would hit the first barrier if the total number of threads is even, and hit the second barrier if the total number of threads is odd.

Textually aligned barrier enables the Titanium compiler to eliminate deadlock bugs due to misaligned barriers at compiler time. Gay and Aiken developed the single inference technique to detect such errors statically [2].

Textually aligned barrier also enables many optimizations in the Titanium compiler, since the set of barriers effectively divides the Titanium program into phases. Analysis such as data race detection can be applied to each phase independently due to the memory consistency model of Titanium. This enables n^2 running time analysis to be scalable in practice, since the number of statements in a phase is typically small. Data prefetch and delayed write optimizations can also be applied to each phase independently of each other, since prefetched reads can not initiate

before the beginning barrier of a phase, and delayed writes must be finished prior to the ending barrier of a phase.

2.4 Single Valued Expressions

Titanium's textually aligned barriers require all threads to execute the same sequence of barriers during program execution. If a barrier appears inside of a branch of a conditional statement, the compiler needs to statically determine that all threads would take the same branch at runtime. This would require the conditional expression to evaluate to the same value on all threads. To aid such analysis at compile time, *single valued expressions* are introduced in Titanium. A single-valued expression evaluates to the same value for all threads. Examples of single valued expressions include constants and Ti.numProcs(), which returns the number of threads running the program. With programmer annotation and compiler inference, the Titanium compiler statically determines which expressions are single-valued. Single valued expressions are used to ensure that barriers line up: a conditional may only contain a barrier if it is guarded by a single-valued expression. The code in Figure 2.4 is erroneous since Ti.thisProc() % 2 == 0 is not single-valued.

2.5 Synchronized Region

Titanium inherits the synchronized region feature from Java. A block of code that is inside of a synchronized region only allows one thread to execute code in the region at a time. Figure 2.5 shows an example use of synchronized region to handle simultaneous updates to a counter.

```
public class Counter {
  private int count = 0;
  public void increment() {
    synchronized (this) {
      count++;
    }
  }
  public int getCount() {
    synchronized (this) {
      return count;
    }
  }
}
```

Figure 2.5: Use of synchronized region for simultaneous counter update

In this example, the counter needs to be incremented simultaneously by different threads. Ordinarily, there would be a risk that two threads could simultaneously try and update the counter at the same time, or one thread is doing an update while another thread is doing a read at the same time. By wrapping the update code in a synchronized region, we avoid this

risk since both the update and the read for the counter require the acquisition of the same lock. Every Java object created has an associated lock or monitor. Putting code inside a synchronized block makes the compiler append instructions to acquire the lock on the specified object before executing the code, and release it afterwards. In Figure 2.5, it is using the lock of the Counter object to guard the synchronized regions for both increment and getCount.

2.6 Titanium Implementation

Figure 2.6 shows the overall structure of the Titanium compiler, which is divided into three components: the Titanium-to-C translator, the Titanium runtime system, and the GASNet communication system. During the first phase of compilation, the Titanium compiler translates Titanium programs into C code in a platform-independent manner, with calls to Titanium-related parallel features such as synchronization, memory access through global pointers, and array copies converted into runtime library calls. The translated C code is next compiled using the target system's C compiler and linked to the runtime system.

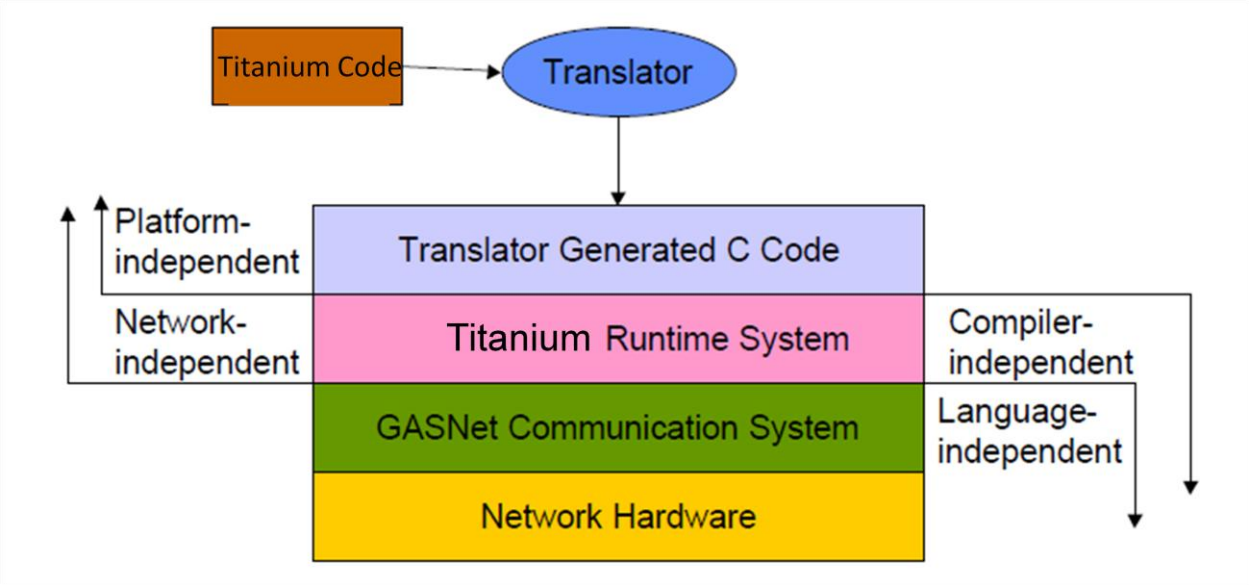


Figure 2.6: Titanium compiler infrastructure

The Titanium runtime delegates communication operations such as remote memory accesses to the GASNet communication layer, which provides a uniform interface for low level communication primitives on all networks. GASNet provides native implementations for several high performance networks including Infiniband, Myrinet, and IBM's LAPI. This three-layer design has several advantages. Because of the choice of C as our intermediate representation, our compiler will be available on most commonly used hardware platforms that have an ISO-compliant C compiler.

In addition to the portability benefits, the layered design also has the goal that each component can be implemented and performance tuned individually. The backend C compiler is free to aggressively optimize the intermediate C output, and the Titanium compiler can utilize its Titanium-specific knowledge to perform communication optimizations. In practice, we have found significant impact on the backend C compiler's ability to do serial optimizations due to the way the Titanium compiler generates C code. For typical serial optimizations applied by the backend C compiler, we have found that the backend C compilers are very sensitive to the way the Titanium compiler generates C code. Substantial efforts we put into the Titanium compiler to allow the backend C compiler to apply serial optimizations. This process is done for each backend C compiler individually. For example, the gcc compiler is more likely to unroll a *for* loop with a constant increment than a *do while* loop with a constant increment. The icc compiler was able to unroll both types of loops. This remains a challenge to Titanium compiler developers on future releases, since the backend C compiler's ability to optimize the generated C code will change over time.

2.7 GASNet

For most Titanium distributed memory backends, the Titanium runtime system delegates communication operations such as remote memory accesses and synchronization to the GASNet communication layer, which provides a uniform interface for low level communication primitives on all networks. GASNet is the communication target for several PGAS languages including Titanum, UPC [67], Co-Array Fortran [52], and Chapel [17]. Unlike MPI, GASNet is designed as a compiler target, rather than an end-user library. It provides Active Messages, one-sided point-to-point messaging, a set of collective operations, and many other features.

The GASNet implementation is designed for both portability and performance. A small set of core functions constitute the basis for portability, and GASNet provides a reference implementation of the full API in terms of this core. In addition, the implementation for a given network can be tuned by implementing any appropriate subset of the general functionality directly upon the hardware-specific primitives, bypassing the reference implementation. The hardware-specific primitives provide higher bandwidth and lower latency primitives.

2.8 GASNet Trace

Titanium's GASNet backends include features that can be used to trace communication using the GASNet trace tool. When a Titanium program is compiled with GASNet trace enabled, a

communication log is kept for each run of the program. In this communication log, each communication event along with the source code line number is recorded. The communication event includes information such as the type of communication (put/get), message size, and the thread ID. GASNet trace is useful in gathering statistics regarding the number of communication calls and the volume of communication. This helps the performance debugging process, but it is not recommended to be used for production runs, since it adds significant overhead on the application performance.

Chapter 3

Irregular Data Access Applications

Irregular data access patterns appear in many scientific applications, including sparse matrix vector multiply, Particle In Cell (PIC) method, histogram, and Giga UPdates per Second (GUPS) [47]. The irregular data access typically contains a data array and an index array, appearing as A[B[i]], where A is the data array and B is the index array. The irregular data access can appear as a read or a write. This dissertation covers both the read and write cases. Irregular write is a harder problem due to the need for synchronization on memory locations that receive updates. This computational pattern is challenging due to the following:

1. The memory locations of the updates are not known until runtime.
2. Updates to the same memory location from different threads must be synchronized.

In this chapter, we present background information on the irregular data access applications that are used throughout this dissertation.

3.1 Histogram

Histogram is a simple benchmark that exemplifies the irregular write data access pattern. In our histogram benchmark, an array of integers are taken as input, each element is mapped to a bucket and counted. The bucket to be incremented is determined based on the value of the integer. The number of dimensions of the target array and the set of elements in the target array receiving updates for a single data point are important characteristics for describing any irregular write algorithm. For the histogram benchmark, the target array is one dimensional, and the update affects a single element of the target array.

We use two different index distributions in our experiments: Web page out-degrees and language encodings. The two index distributions differ in the number of buckets and the clustering of the data points. In the language encodings distribution, there are around 100 language encodings widely in used in the World Wide Web. Each bucket represents one language encoding, therefore making about 100 buckets in the histogram. Figure 3.1 shows the language encoding distribution.

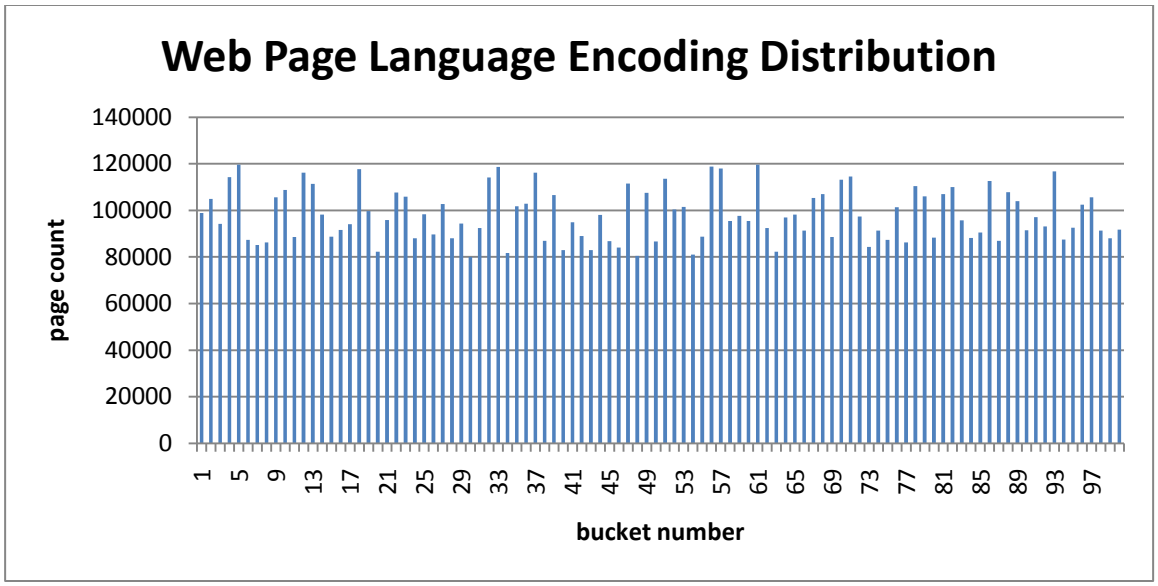


Figure 3.1: Web page language encoding distribution

In comparison, there are many more buckets in the Web page out-degree distribution, where each out-degree count represents one bucket. The range of out-degrees in the dataset ranges from 0 to 40000. Most human written Web pages have fewer than 100 outlinks. This characteristic causes clustering for buckets in the range of 0 to 100. The chances of collision in the two distributions are similar, since they have roughly 100 buckets that receive updates. The difference is the Web page out-degree distribution has many more buckets, where most of them receive zero updates. Figure 3.2 shows the Web page out-degree distribution.

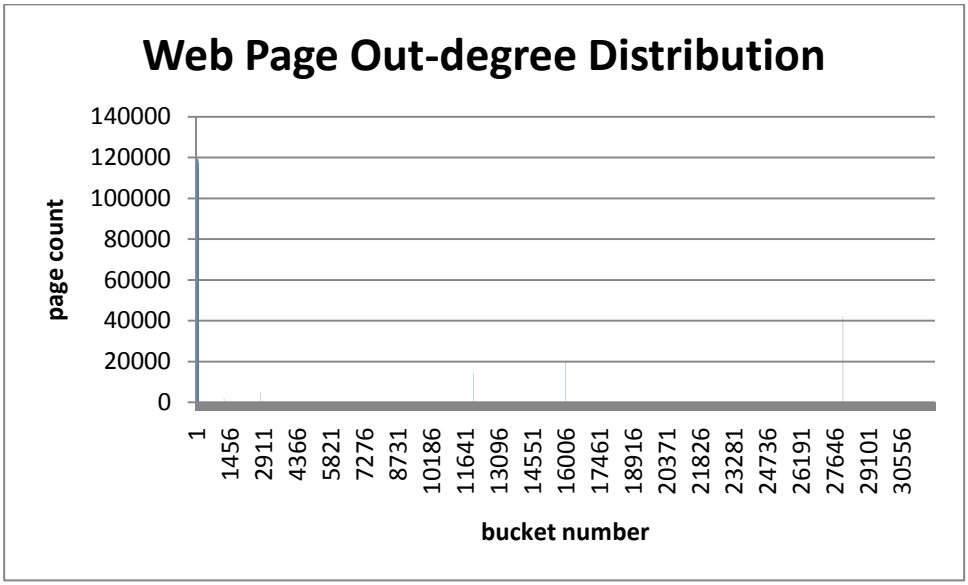


Figure 3.2: Web page out-degree distribution

3.2 Particle Gravitation

In the particle gravitation benchmark, particles are spread around in 3D space. Each particle exerts a gravitational force on every other particle. Particles are partitioned evenly among processors. As processors work on its set of particles, two processors may update the force field of a particle simultaneously. Synchronization is required to ensure atomic update to the force field. Particles are represented as objects, so that an update to its force field requires a level of indirection.

3.3 Adaptive Mesh Refinement

Adaptive Mesh Refinement is one of the largest Titanium applications. It is used in the performance debugging chapter of this dissertation. The AMR methodology has been successfully applied to numerical modeling of various physical problems that exhibit multiscale behavior, such as those mentioned in. The idea of AMR is quite straightforward, that is, to apply finer discretization only at places where higher resolution is needed. However, the simplicity of finite difference calculation on a uniform grid is traded in AMR, where the irregularity comes from the boundaries between grids introduced by local mesh refinement. Figure 3.3 illustrates the interaction between the fine grids and their corresponding coarse grids. Irregular computations arise in updating boundary values, for example, the location-dependent quadratic interpolation of ghost values at the coarse-fine grid interface.

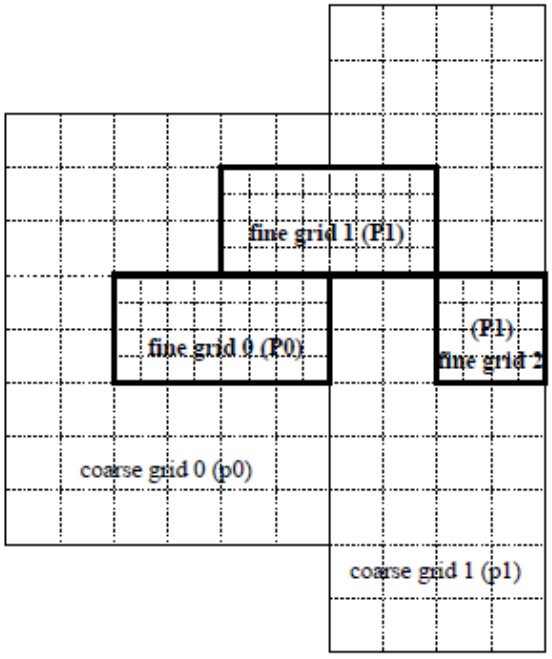


Figure 3.3: Interactions between fine grid and coarse grid in AMR

3.4 Immersed Boundary Method

One of the benchmarks used throughout this dissertation is the spread force operation in the Heart simulation code. The Heart simulation code uses the immersed boundary method [51] to simulate blood flow through the heart. The immersed boundary method is a general technique for modeling elastic boundaries immersed within a viscous, incompressible fluid. The method has been applied to several biological and engineering systems, including the simulation of blood flow in the heart, sound waves in the cochlea, and clotting of blood. These simulations have the potential to improve our basic understanding of the biological systems they model and aid in the development of surgical treatments and prosthetic devices. Despite the popularity of the immersed boundary method and the desire to scale the problems to accurately capture the details of the physical systems, parallelization for large scale distributed memory machine has proven challenging. Figure 3.4 shows the four phases of the Heart simulation.

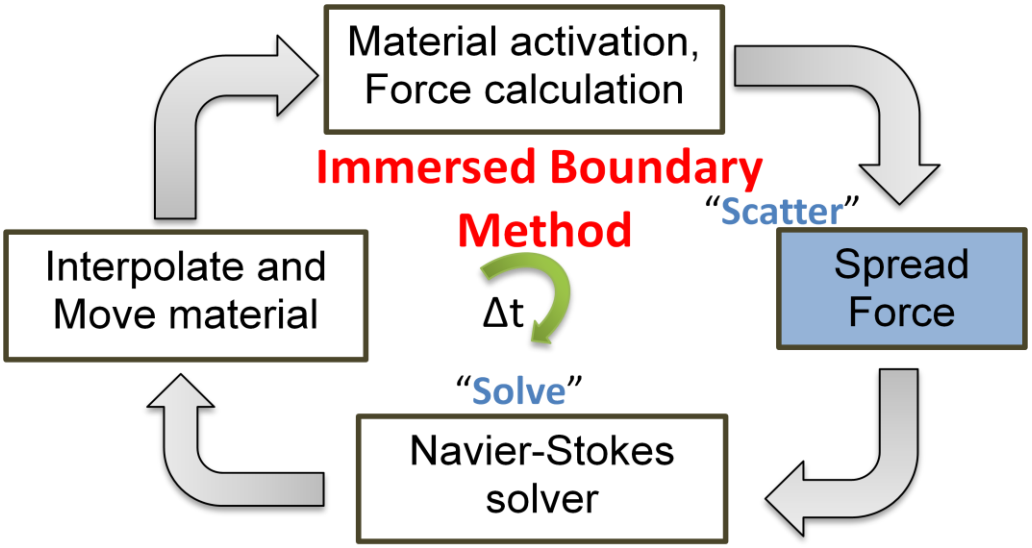


Figure 3.4: The four phases of the immersed boundary method

One of the most challenging operations in the heart simulation is spread force, where the forces from the heart fiber point are spread to the neighboring fluid cells. The target array is 3D, and the particle spreads force to its nearest 4x4x4 cube of fluid cells. In this dissertation, we explore optimizations that are applicable to the spread force operation on both multicore processors and clusters.

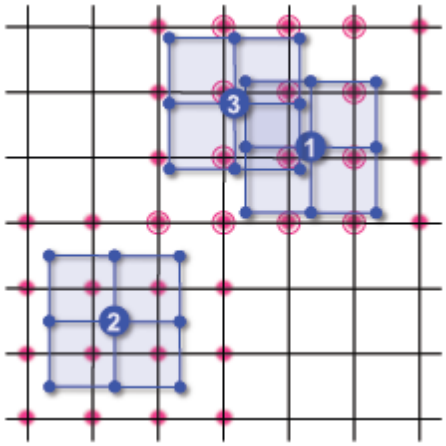


Figure 3.5: 2D Heart spread force. Each particle affects its nearby 4x4 neighbor patch

In Figure 3.5, we visualize a 2D particle-to-grid spread force operation of the Heart Code. As the patch is interpolated into the grid, 16 point must be updated (the 3D Heart Code updates 64 points in a 4x4x4 cube). Enlarged circles represent grid points that receive updates from multiple patches. Proper synchronization must be used to avoid data races on those grid points. For example, there are two grid points in the upper right corner of patch 2 that overlap with patch 1 and patch 3. Between patch 1 and patch 3, there are 12 grid points that overlap.

The density of heart fiber points is not uniform throughout the fluid grid. In the beginning of the simulation, the heart fiber points are grouped together in a heart shape at the center of the fluid cell. As the simulation progresses, the fiber points slowly drift away from the center. High density of fiber points are found at the center of the fluid grid. Regions away from the center have sparse population of fiber points. For the experiments in this chapter, we use a sphere instead of the actual heart for simulation. The source code is the same, only the initial positions of the fiber points differ to make up a different shape. It is much easier to generate inputs for different size problems for the sphere. For the heart, we only have inputs for a single configuration from clinical data. The sizes of the fluid grid used in our experiments are 128^3, 192^3, and 256^3.

There are two input parameters in the heart code to describe the test simulation. They are the grid size and number of particles. We vary the number of particles and the grid size in our experiments. The particles form a sphere centered in the fluid grid. Particles are evenly distributed on the surface of the sphere. By varying the number of particles, we change the particle density on the surface of the sphere. By varying the grid size, we change the memory footprint of the problem instance, which is analogous to increasing the number of buckets in a histogram.

3.5 Sparse Matrix Vector Multiply

Indirect array accesses and the irregular memory access patterns that result are common in sparse matrix code. In this dissertation, we use a sparse matrix kernel, matrix vector multiplication, to evaluate our compiler and runtime techniques for optimizing irregular read data access patterns. In this case, the matrix is sparse while both the source and result vectors are dense. The parallel algorithm partitions the matrix by rows, with each processor getting a contiguous block of complete rows. Each processor also holds the corresponding piece of the result vector, so the only communication that is required is on the source vector. Because the source vector is often computed from an earlier result, it is partitioned in the same manner as the result vector. Figure 3.6 illustrates the layout of the matrix in the case with eight processors. Communication is only required for the source vector, and only for those elements in which a processor holds a nonzero outside of the processor's diagonal block. The off-diagonal nonzero shown will result in communication from P5 to P1, for example.

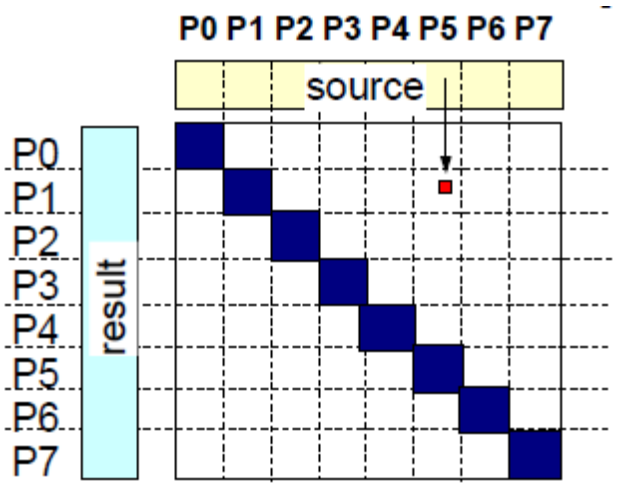


Figure 3.6: Parallel layouts of matrix and vector

Chapter 4

Optimizing Irregular Reads

In this chapter, we present compiler and runtime extensions for the Titanium implementation to support programs with indirect read array accesses, such as A[B[i]], where some elements of the A array may live in a remote processor's memory. The effectiveness of the optimizations is evaluated on three different cluster architectures. These computations arise in sparse iterative solvers, particle-mesh methods, and elsewhere. We add compiler support for an *inspector executor* execution model, which optimizes communication performing runtime optimization based on the dynamic pattern of indices in the indirection array, which is B in the previous example. There are several possible transformations that can be done on the communication, and we consider three in this chapter: sending the entire remote array, sending exactly those elements that are needed, and sending a bounding box of the required values. While packing is guaranteed to send the minimal number of actual values, it has a higher metadata overhead and is therefore not necessarily optimal. One of the challenges is to select the best communication transformation, because it depends on properties of the application and the machine. We introduce a simple analytical performance model into the compiler, which selects optimizations automatically. We analyze the benefits of the automated *inspector-executor* transformation using sparse matrix vector multiplication on a set of matrices from real applications. Our results show that although the program is significantly simpler when written in Titanium, because it avoids the explicit communication code and the pack and unpack code, the performance is almost always superior to a popular MPI message passing code. The speedup relative to MPI on a suite of over 20 matrices averages 21% on three different machines, with the maximum speedup of more than 2x.

The model-based optimization selection is critical to both programmability and performance. Not only does the model select optimizations that differ by matrix and machine, but also differ between processors within a single matrix-vector multiplication. This runtime complexity is entirely hidden from the programmer, making the application both cleaner and faster.

4.1 Source Code Transformation

In this section, we give motivations for the optimizations later in the chapter using source code transformation on a simple example. The simple example is the indirect sum benchmark in Figure 4.1.

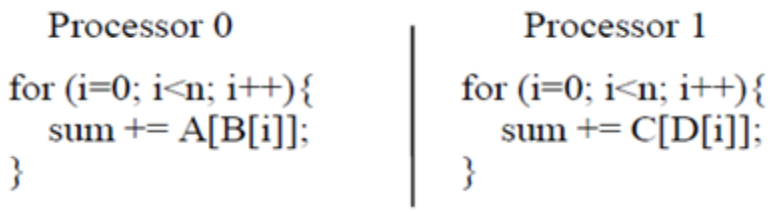


Figure 4.1: Indirect sum benchmark on two processors

The example illustrates the case for two processors, but can easily be extended for more processors. A is an array that resides on processor 1, and B is an index array owned by processor 0 for accessing array A. When processor 0 reads A[B[i]], it requires communication, because A resides on processor 1. Analogously, C is an array on processor 0, and D is an index array owned by processor 1 for accessing array C.

In Figure 4.2, we introduce buffers into the indirect sum benchmark. The buffers are local to each processor. The values of each indirect array access are stored in the buffer, and subsequently used in the second loop. This transformation is legal because neither A or B appear in the left-hand side of an assignment statement in the code on either processor, so it changes neither the local dependence order nor the order of any conflicting accesses across processors.

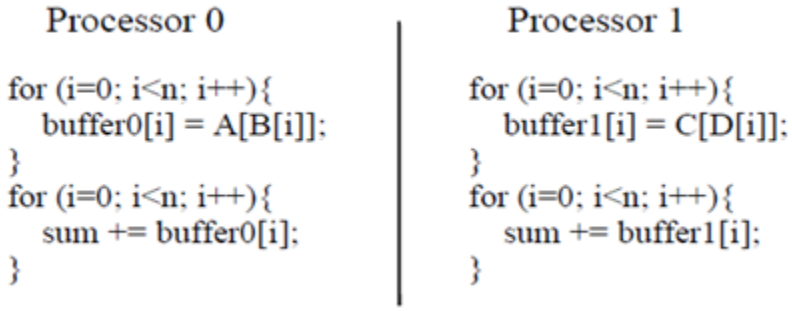


Figure 4.2: Indirect sum benchmark on two processors with local buffers

In Figure 4.1 and Figure 4.2, the communication uses a pull strategy. Each processor gets the data it needs from the remote processor that owns the data. The next source code transformation in Figure 4.3 uses a push strategy instead. This code does not show the entire transformation, which includes recording information about which values each processor will need from the other and the location of remote buffers in which to store the data. This data is computed in a separate preprocessing phase and is performed only once, whereas the code in

Figure 4.3 will be executed many times, e.g., one per iteration of a solver. In the push strategy, data is put to the processor that needs it.

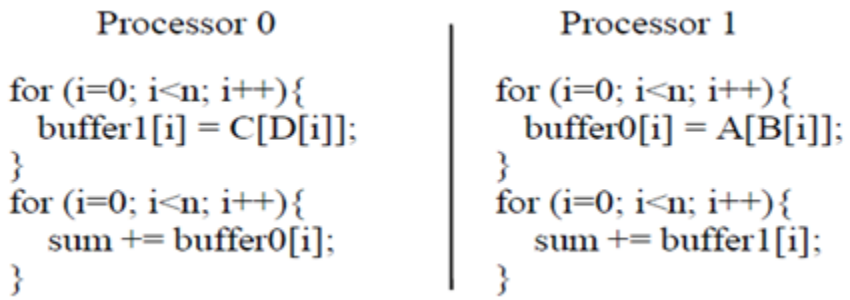


Figure 4.3: Indirect sum benchmark on two processors using push strategy

Two questions arise from these source code transformations. The first question is when is it legal to apply these transformations. The second question is how do these transformations give us the desired speedup. These questions will be addressed later in this chapter.

4.2 Compile Time Transformations

4.2.1 Identify Inspector Executor Candidates

The first step is to identify *inspector executor* candidates for indirect array accesses A[B[i]]. Below is the list of the conditions that the compiler checks for:

- A, B, and all of the elements A[i] and B[i] are not modified either directly or through aliases by the processor reading them. This ensures local dependencies are preserved.
- There are no synchronization points inside of the foreach loop, since the memory model requires memory to be globally consistent at a synchronization point.

The memory consistency model in Titanium does not require a check for conflicting accesses across threads, so the program behavior could be surprising if A and B were modified by another thread. After identifying the candidates, the compiler performs the *inspector executor* transformation. In the inspector phase, the array address for each A[B[i]] is computed. The computed values are stored in an index array. After the inspector phase, a communication method is chosen to retrieve the remote data into a local buffer. More details on the choice of the communication method are presented in Section 4.4. The set of array addresses together with the communication method are stored in a communication schedule. In the executor loop, values for each A[B[i]] are read out of the local buffer.

In some applications, the same pattern of indirect array accesses happens over multiple iterations. One example is an iterative solver. In this case, we would like to store the communication schedule computed during the inspector phase of the first iteration, and reuse the communication schedule on other iterations. A communication schedule may contain information for one or more sets of indirect array accesses to remote arrays. For each set of array accesses, the computed array addresses and the choice of communication method are stored in the schedule. Schedule reuse has been used in prior work, but our schedules contain additional information about the communication method to be employed.

The three properties are sufficient for ensuring the soundness of the *inspector executor* transformation. First we show that the values read for B[i] are the same for both versions of the program. If B and B[i] do not change during the execution of the loop, then the values read for B[i] in the inspector are the same as the ones in the original loop. B can only change locally, because it is a pointer on the local processor. The changes to B[i] can either come locally or remotely. We know that there are no local changes, because we check that B and B[i] are loop invariant using defuse information. Remote changes to B[i] are possible, since any of the remote processors can be executing code that modifies B[i] while the local processor is inside the loop. If we take a snap shot of memory where the B[i]'s reside before the local processor enters the loop, we can use the values from the snap shot for B[i]'s inside the loop regardless of whether there are changes to B[i] remotely. The reason is that there are no synchronization events inside the loop, so any remote changes to B[i] during the execution of the loop do not need to be reflected under the Titanium memory consistency model. Any remote writes to B[i] are allowed to be reordered relative to local reads and writes, so they may all appear to happen after the loop completes.

Now we know the index sets for both versions are the same. We would like to show that the values read from A using this index set are the same for both versions of the program. The argument is similar to the previous step. We know that there are no local changes to A or A[i] using defuse information. Changes to A[i] caused by remote processors during the loop do not need to be reflected, because there is no synchronization event inside the loop.

The requirement that A[i] is not modified by the processor executing the loop can be relaxed. The processor executing the loop has access to the buffer that contains the prefetched values of A[B[i]]. The runtime can conceivably intercept all writes to A[i] from this processor in the loop, and reflect the changes to the values in the buffer. Our experiments show that this relaxation is not worthwhile.

4.2.2 Pull to Push Transformation

Now we know the requirements for applying the *inspector executor* transformation. In this section, we turn our attention to the issue of using the push strategy instead of the pull strategy for communication. Extra coordination between processors is needed to use the push strategy. In the case where the schedule can be reused, we would like to communicate the index set and the choice of communication method during the first iteration, and have the processor that owns the data to send the needed data in the subsequent iterations independently. The push strategy uses about half as many messages as the pull strategy. The pull strategy also suffers when the remote processor is not attentive to the network. The communication is not entirely one-sided, because remote packing is required, so if the remote processor is in a computation intensive loop, other processors may be delayed waiting for it.

The extra coordination means that the communication calls need to be placed in the right place so that the data will be coming when it is expected. It is the job of the compiler to find the right place in the code to insert these communication calls, since we are applying the optimizations automatically without hints from the application programmer.

The communication calls need to be placed in such a way that when a processor is about to enter the loop that contains the *inspector executor* array access, the expected data is on its way from the remote processor. The processor can simply poll on a flag for the arrival of the data.

In Titanium, a barrier statement is executed by all the processors at the same time for the same number of times. Unlike other languages and libraries, the simultaneous barriers must be the same textual instance in the program, so a barrier in one branch of conditional cannot match a barrier in another. This prevents barrier deadlocks in which a barrier is executed by only a subset of the processors and therefore deadlocks. The Titanium compiler provides a static analysis called *single analysis* that eliminates this type of bugs. It conservatively rejects all programs that might run into deadlocks due to misplaced barriers at compile time.

We use *single analysis* to help us in finding the right place to insert the communication calls. The property that we are looking for is that whenever a processor is about to enter the loop containing the *inspector executor* array access, the processor that owns the needed data would execute the communication calls to send the data over. In the top of the loop that contains the *inspector executor* array access, we insert a barrier node in the control flow graph. Then we run single analysis on this modified control flow graph. Single analysis tells us if it is safe to place the barrier in the top of the loop. If it is not safe, then it is not safe to place the communication calls there, because the processor that needs the data and the processor that owns the data may come to the top of the loop at different times or for different number of times. If single analysis tells us that it is safe to place a barrier in the top of the loop, then we can place the communication calls there, because we are certain that all processors would come to the top of

the loop around the same time for the same number of times. After running single analysis, we remove the barrier node from the control flow graph.

4.2.3 Overlap

Our generated code utilizes two types of overlap: communication with communication, and communication with computation. When a processor owns data that is needed by multiple remote processors, non-blocking puts are used to push the needed data to the remote processors. While waiting for the remote data to arrive, we can overlap the computation that only involves local elements or computation with elements that have already arrived.

4.3 Experimental Platforms

We performed experiments on three parallel machines and developed a performance model for the communication on each of them. This includes a cluster of Itanium processors connected by a Myrinet network, an IBM Power3 system, and an HP system with a Quandrics interconnect. Table 4.1 contains a summary of their key features.

Name	System	Network	CPU
RTC	Linux cluster	Myrinet 2000	900 MHz Itanium 2
Seaborg	IBM RS/6000 SP	SP Colony Switch 2	375 MHz Power 3+
Lemieux	Compaq Alphaserver ES45	Quadrics Elan3	1 GHz Alpha

Table 4.1: machine summary

4.4 Runtime Selection of Communication

With a set of indirect array accesses to a remote array, there are several options for performing the data communication. The options are listed below:

- *Pack method*: only communicates the needed elements without duplicates. The needed elements are packed into a buffer before sending them to the processor that needs the data.

- *Bound method*: use a bulk put operation to send a bounding box that contains the needed elements.
- *Bulk method*: use a bulk put operation to send the entire array.

The three methods require different amount of set up work. The pack method needs to run the inspector phase to translate all the indirect array accesses into remote addresses and the bound method needs to run the inspector phase to compute the bounding box that contains all the needed elements. The bulk method does not require an inspector phase.

In this chapter, we focus on the case where the communication pattern is repeated several times, in which case the cost of the inspector is amortized, so we always run the inspector in the first iteration. In this scenario the bulk method becomes a special case of the bounding box method, so we only discuss the pack and bound methods in the remainder of this chapter.

Our experiments show that the choice of communication is both application and machine specific. The application determines the size of the array, number of accesses to that array, and the size of the bounding box. The machine gives different memory and communication costs. Our compiler generates code that can choose the best communication method at runtime based on a performance model.

The total time of a communication method consists of three parts:

1. the time spent on getting the data ready for communication in the remote processor
2. the communication time for sending the data
3. the time for reading the data out of local buffer in the executor loop

Both 1 and 3 are local processor costs, which are dominated by memory access times. We use cache and memory latency numbers provided by the vendor in a performance model. In the pack method, the needed elements are gathered into a buffer by the remote processor. The gathering from the source array is random access, while the storing of the elements into the buffer and the reading of indices are sequential access. In the bound method, no packing of the data is needed, since the entire bounding box is sent. Figure 4.4 shows the model components. N is the number of distinct elements being packed, $L1_{line}$ and $L2_{line}$ are cache line sizes, and α_1, α_2 and α_{mem} are latencies for L1, L2, and memory, respectively. Cache line sizes are adjusted to match the word size in each formula.

buffer gather:

$N\alpha_1 + N / (L1_{line})(\alpha_2 - \alpha_1) + N / (L2_{line})*(\alpha_{mem} - \alpha_2)$

index read:

$N\alpha_1 + N / (L1_{line})(\alpha_2 - \alpha_1) + N / (L2_{line})*(\alpha_{mem} - \alpha_2)$

source:

if (source fits in L1) then $N\alpha_1$
else if (source fits in L2) then $N\alpha_2$
else $N\alpha_{mem}$

Figure 4.4: Performance model for packing

To estimate the communication cost, we use a piecewise linear model. For large messages, the cost is a fixed per message latency plus a per Byte bandwidth cost. We found that using this simple linear model for small messages was not accurate enough, and instead use different latency and bandwidth terms in different size ranges. For example, in the GM network, the steps are due to the 4KB MTU size of the packets. Figure 4.5 shows how well our models fit the actual. The average error comparing our models with the actual on all three machines is less than 1%. These latency and bandwidth numbers are computed empirically for each machine.

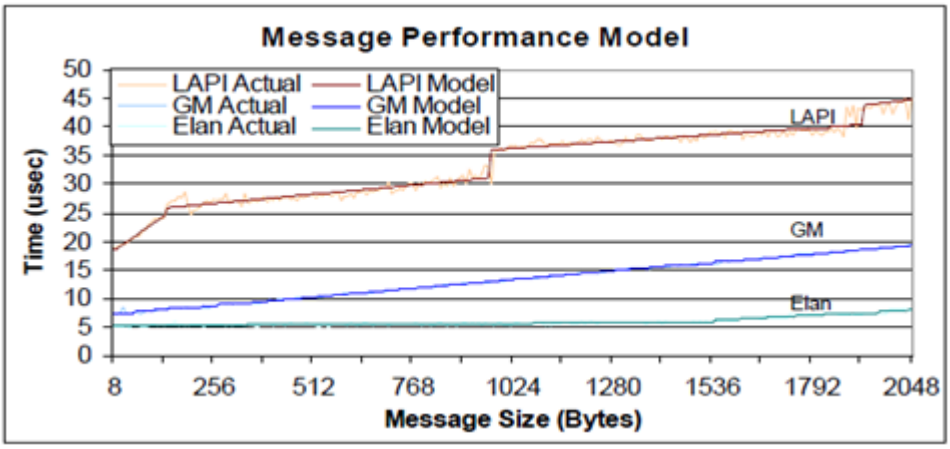


Figure 4.5: Comparing the latency bandwidth models to actual on LAPI, GM, and Elan

In practice, each processor may communicate with multiple other processors, while our model is for a single pair of processors and does not account for network contention. The number of simultaneous communication events depends on the application characteristics. While our point to point model does not capture this more complex communication behavior, we find it is

sufficient for selecting a good communication method, as we will show in the performance section.

After the data arrives at the destination processor, the data is read out of the buffer during the executor phase in a random access pattern. In our model, we assume the costs for both methods are the same, although the bound method may suffer more cache misses in practice, because it is a larger data buffer than in the pack case.

We pick the communication method that gives the lowest estimated cost by adding the packing and communication cost estimates. A choice is made separately for each processor pair. A schedule may contain several pairs, since each processor may need to communicate with several remote processors. The communication method selection automatically trades off network bandwidth and cache misses. By packing the needed elements into a buffer, the pack method uses a smaller message than the bounding box, but it incurs more memory traffic for the packing process.

4.5 Optimizing a Sparse Matrix Kernel

Indirect array accesses and the irregular memory and network access patterns that result are common in sparse matrix code. In this section we use a sparse matrix kernel, matrix vector multiplication, to evaluate our compiler and runtime techniques, which are entirely automatic. In this case, the matrix is sparse while both the source and result vectors are dense.

The parallel algorithm partitions the matrix by rows, with each processor getting a contiguous block of complete rows. Each processor also holds the corresponding piece of the result vector, so the only communication that is required is on the source vector. Because the source vector is often computed from an earlier result, it is partitioned in the same manner as the result vector. Figure 4.6 illustrates the layout of the matrix in the case with eight processors. Communication is only required for the source vector, and only for those elements in which a processor holds a nonzero outside of the processor's diagonal block. The off-diagonal nonzero shown will result in communication from P5 to P1, for example.

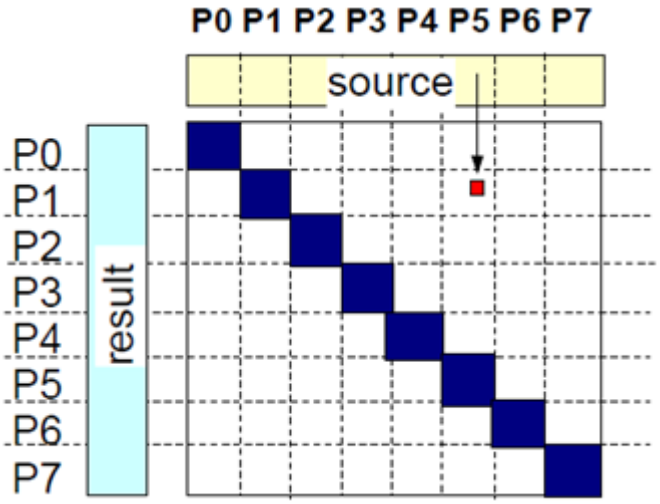


Figure 4.6: Parallel layouts of matrix and vectors

Due to the different nonzero structures of the matrices, the communication requirements vary widely across matrices. We therefore use a set of benchmark matrices from real applications to evaluate our optimizations. Figure 4.7 gives two examples to illustrate the differences. On the left we have the nemeth21 from Matrix Market [49]. It is a 9506x9506 matrix with 1173746 nonzeros. Because the nonzero occur only near the diagonal, each processor needs to communicate with at most two of its neighbors. The garon2 matrix on the right is taken from the UF Sparse Matrix Collection and is a 2D finite element method matrix [66]. The size is 13535x13535 and there are 390607 nonzeros. There is more data communication for this matrix than the previous one, because nonzero are spread throughout the matrix, albeit in a regular pattern. Every processor will need some data from every other processor.

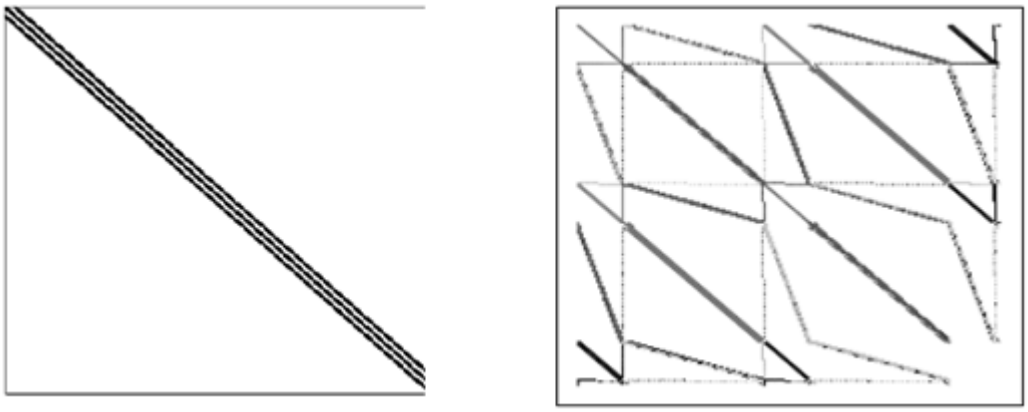


Figure 4.7: Structure of nemeth21 (left) and garon2 (right)

4.6 Evaluating Each Optimization

We begin by analyzing the performance of several different Titanium implementations using the garon2 matrix, which will highlight the differences in communication costs between the versions. The Titanium source code is the same across the versions, but the compiler and runtime support differ. Figure 4.8 shows the performance on the Itanium/Myrinet cluster using the GASNet implementation for Myrinet's GM1 communication layer.

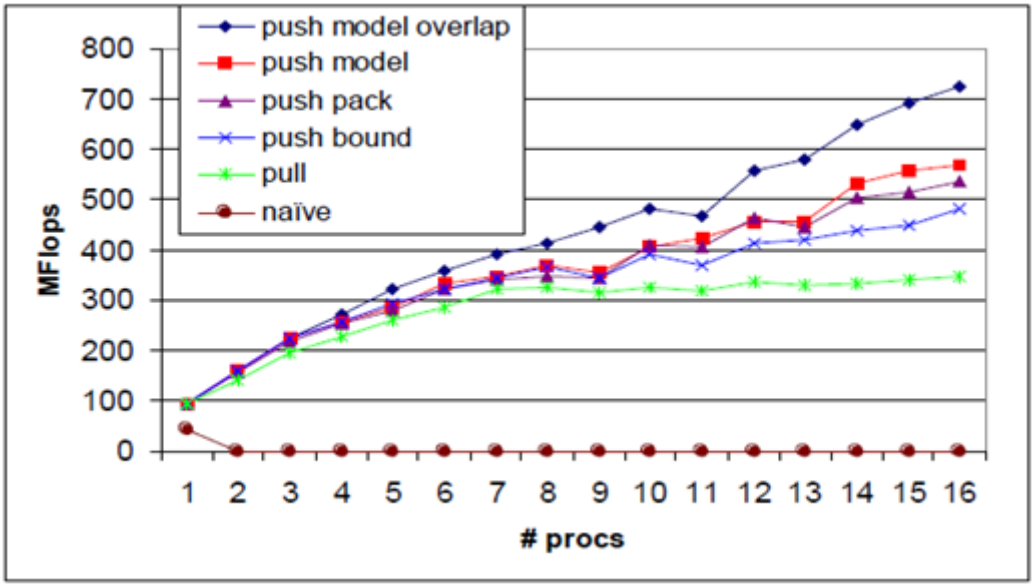


Figure 4.8: Performance on the garon2 matrix

Naïve: With the naïve version, the generated code uses a remote read for each indirect array access when the array is remote. As expected, the performance does not scale.

Pull: The next line shows the benefit of an initial inspector executor optimization on this code. This significantly improves on the naïve version, because data is packed into larger messages. Remote gets are used in this case to pull data from the remote parts of the source vector.

Push: The next three versions of the code uses remote puts instead of gets for communication. On Myrinet GM1, puts are significantly faster than gets because puts are implemented using lower level RDMA support. Pulling requires a round-trip with remote work to pack the data, which requires that all processors be attentive to the network. The service of the packing request can be delayed if the remote processor is in the middle of a computation intensive loop when the request arrives.

Model vs Pure: The three push lines differ in the choice of communication mechanism. One uses the bounding box approach throughout the machine, while another uses packing throughout. The third line uses our performance model, which performs at least as well as the best of the pure methods. For some processor configurations, the model actually chooses a mixed strategy, such as in the 16 processors case. In a mixed strategy, a processor communicates with some neighbors using the bound method, and other neighbors with the pack method. This is the reason that we see the gap between the model performance and the maximum of the pure method performances. This result shows that applying the inspector executor optimization manually at the source code level can be a daunting task. Because there is a choice of communication methods between each pair of processors, the number of different choice configurations grows exponentially as the number of neighbor increases. Furthermore, the search for the best configuration would have to be done for each combination of input and processor configuration. We believe that it is a much cleaner solution to develop a performance model, and have the compiler to apply the inspector executor optimization automatically. The performance model only has to be developed once for a given machine.

Overlap: The last version of the code overlaps communication with computation. After sending the needed data using non-blocking puts, each processor does its local computation with its nonzeros that do not require communication. After the local computation completes, the processor polls on the arrival of the remote data as they are needed. On some matrices, there are sufficiently many nonzeros that do not require communication that by the time the processor needs the incoming data, the data has already arrived. In those cases, the cost of communication is largely hidden behind the local computation.

4.7 Comparison with an MPI Library

In this section we compare the best Titanium implementation using the performance model and overlapped communication with an MPI implementation that uses the same basic data layout and algorithm for sparse matrix-vector multiplication. The MPI implementation is from a popular sparse solver library called Aztec, which is written in C [65].

We use matrices from Matrix Market and the UF Sparse Matrix Collection. Table 4.2 lists some of the matrices along with the dimensionality (all the matrices are square) and the number of nonzeros.

#	Name	Dim (NxN)	Non-zeros	Area
1	appu	14000	1853104	Structures
2	av41092	41092	1683902	Unknown
3	barrier2-1	113076	2129496	Physics
4	bbmat	38744	1771722	Structures
5	cage12	130228	2032536	Biology
6	cfd2	123440	3085406	Graphics
7	crystk02	13965	968583	Structures
8	crystk03	24696	1751178	Structures
9	lin	256000	1766400	Eigenval
10	nasasrb	54870	2677324	Structures
11	nemeth21	9506	1173746	Chemistry
12	nemeth22	9506	1358832	Chemistry
13	nemeth23	9506	1506810	Chemistry
14	nemeth24	9506	1506550	Chemistry
15	nemeth25	9506	1511758	Chemistry
16	nemeth26	9506	1511760	Chemistry
17	oilpan	73752	2148558	FEM
18	qa8fk	66127	1660579	FEM
19	qa8fm	66127	1660579	FEM
20	t3dh_a	79171	4352105	Physics
21	t3dh_e	79171	4352105	Physics
22	vanbody	47072	2329056	FEM

Table 4.2: Matrix characteristics

Programmability differences between C+MPI and Titanium are difficult to quantify. The Titanium code contains only array accesses and field dereferences, whereas the MPI code has explicit send and receive routines as well as code to pack required source vector elements into buffers. Lines of source code, while far from perfect, may provide some insight into the differences in programming difficulty. The C program using Aztec is 55% longer than the Titanium code.

We performed experiments on the three machines described earlier. The Titanium implementation uses tuned GASNet implementations for each of the three message layers: GM1 on Myrinet, Elan3 on Quadrics, and LAPI on the IBM SP. Aztec uses a pure packing approach for communication. We use different processor configurations from 1 to 16 processors, and always use only one processor per node.

Figures 4.9 through 4.11 show the average and maximum speedup of the Titanium version relative to the Aztec version on 1 to 16 processors. In general, the Titanium version is faster, sometimes by more than a factor of 2x. Across all processor configurations, matrices, and machine, the Titanium code was an average of 1.2x faster than the MPI code. The speedups are highest on the Myrinet machine, where the RDMA support used by GASnet is most significant. The Quadrics network is fast for both MPI and GASNet, and for matrices with less communication, you see little difference between the two languages. There are some cases where the MPI code outperforms the Titanium code, usually on smaller problem sizes. We continue to investigate issues related to barrier performance and serial code differences that account for these slowdowns, and presumable additional tuning for these machines may also be possible in the Aztec code.

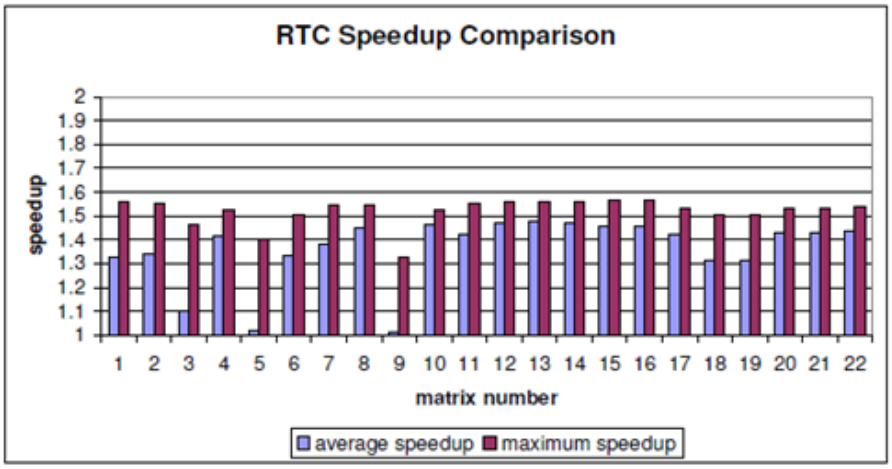


Figure 4.9: Performance comparison between Titanium and Aztec on Linux cluster

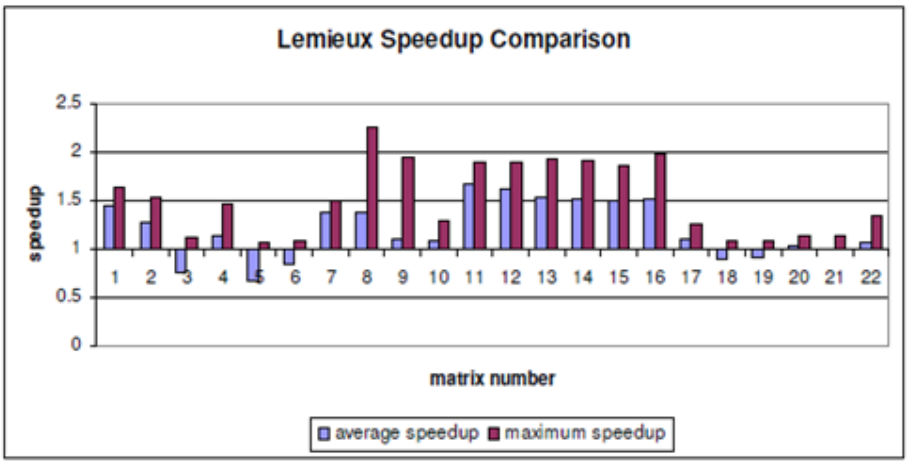


Figure 4.10: Performance comparison between Titanium and Aztec on Compaq Alphaserver

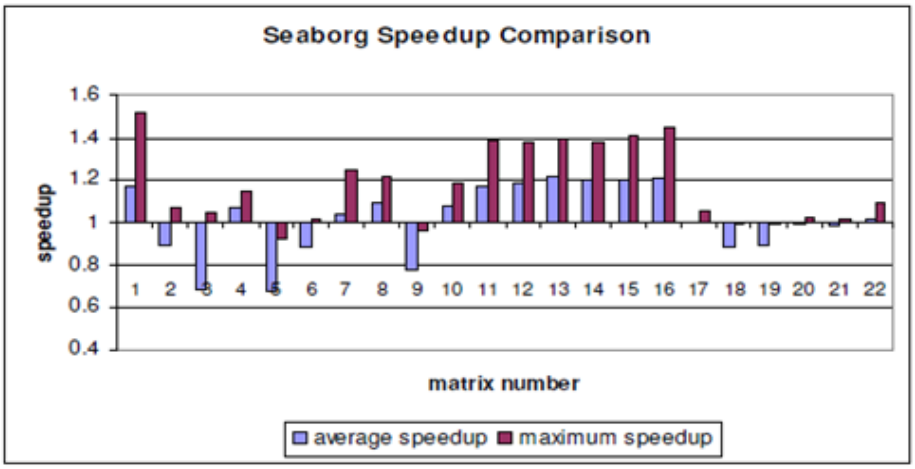


Figure 4.11: Performance comparison between Titanium and Aztec on IBM SP

Our overall summary is that the performance of the Titanium code is usually better than that of MPI and from the analysis in section 4.7, we can attribute these differences to the combination of the RDMA support in GASNet, avoiding the cost of packing for some matrices, and the use of a performance model to select the best communication mechanism for each pair of processors, and the effectiveness of the overlap using the RDMA communication model.

4.8 Conclusions

In this chapter, we have described an automatic transformation of programs with indirect array accesses in Titanium to take advantage of inspector executor style optimizations. We introduced a performance modeling technique to select communication methods using a combination of data collected at compiler install time and runtime information about the application's access patterns. This allows application programmers to write Titanium code in a straightforward way and obtain performance superior to a popular hand-tuned library. In particular, for a sparse matrix vector multiply problem, we showed that not only is the program more concise, but the optimized Titanium code is up to 2.25x faster.

These results show the feasibility of using a high level language for high performance scientific programming on programs with indirect array accesses on remote arrays. In other papers we have demonstrated the use of Titanium on large applications, including a heart simulation and adaptive mesh refinement problems. The use of a high level parallel language enables compiler optimizations, like the one described in this chapter, that cannot be done automatically in a library-based approach like MPI.

Chapter 5

Optimizing for Irregular Writes

5.1 Introduction

In the previous chapter, we describe different optimization techniques for irregular updates on multicore processors. Applications with irregular updates also post a significant challenge for distributed memory backends. In this chapter, we present analysis and optimization techniques for this challenge in the context of Titanium. The optimization goals for both the multicore and cluster case are similar: we would like to maximize the number of local updates while minimizing the cost for remote updates. As seen in the previous chapter, local updates on multicore occur on memory that is mapped on the same socket. Remote updates go to memory that is mapped to a separate socket. The memory latency and bandwidth differences between these two types of accesses shape our optimization goal. On clusters, the performance difference is even more dramatic. Local updates occur on shared memory within the same node. Remote updates require network roundtrip to update memory on a remote machine. We explore a similar set of optimization techniques such as locking and update by owner in the cluster setting.

Our optimization work on clusters is done in the context of Titanium. Synchronized regions are often needed to write fine grain parallel programs. This includes applications such as particle in cell methods, where irregular updates to fluid cells from different processors need to be synchronized. Properly written synchronized regions allow atomic updates and accesses to shared data. A straightforward implementation of synchronized regions requires the use of global locks. Global locking is a very expensive operation, especially in a distributed memory environment. A network roundtrip is needed for both the lock acquisition and lock release.

In this chapter, we describe an automatic compiler optimization for synchronized regions that can eliminate the need for global locks entirely in cases where update by owner is sufficient. We use the term update by owner to mean having the processor that owns the underlying data to process updates to this data. We used three benchmarks including the spread force operator in the heart simulation application, histogram, and particle gravitation to evaluate the effectiveness of the optimization. On a cluster of Xeons, the optimization yields speedups up to three orders of magnitude over the same code compiled without the optimization.

5.2 Motivating Example

For particle mesh method codes written for shared memory, synchronized regions are often used to synchronize updates from different processors to the same mesh cell. Below is an example use of synchronized regions in a particle in cell code for spreading the force from particles to its neighboring fluid cells. Since a fluid cell may have several neighboring particles, updates to the fluid cell must be synchronized.

The programmer may choose to use different locking granularity for this problem. Figure 5.1 and Figure 5.2 illustrate locking at the global fluid grid level and fluid cell level, respectively. Each processor goes through the particles it owns, and spread the particle's force to the neighboring fluid cells. A processor can lock the globalFluidLock once for the entire fluid grid while doing all of its updates. This approach has less locking overhead, since each processor only enters the synchronized region once. But it effectively serializes all updates from the different processors. Locking at the fluid cell level on each update, it affords the most concurrency, since updates to different cells can happen concurrently. But the locking overhead is prohibitively expensive: each update requires a lock and unlock. The programmer may choose to write the program in at any granularity level. The optimization technique described in this chapter can be applied to both versions.

```
synchronized (globalFluidLock) {
    foreach (index in myParticleArray.domain()) {
        Particle par = myParticleArray[index];
        foreach (pt in neighbor(par)) {
            update(fluid, pt, par);
        }
    }
}
```

Figure 5.1 Spread force operation using a single lock for the entire fluid grid. The fluid array is shared by all processors.

```
foreach (index in myParticleArray.domain()) {
    Particle par = myParticleArray[index];
    foreach (pt in neighbor(par)) {
      Lock lock = lockPerCellArray[pt];
      synchronized (lock) {
        update(fluid, pt, par);
      }
    }
}
```

Figure 5.2 Spread force operation using one lock per fluid cell.

Application programmers using Titanium may resort to manual optimization techniques to do away with the lock. For this example, the manual optimization has each processor cache its own updates to the fluid cells. The cache fluid array is initially set to zero. When it has gone through all the particles, the aggregated update values are stored in the cache fluid array. It then sends the cache fluid array to the processor that owns the corresponding fluid cell. Each processor then processes the incoming updates. We characterize this type of optimization as update by owner. This optimization eliminates the need for global locks entirely. Because the processor that owns the underlying fluid cell processes the updates sequentially, this effectively synchronizes the updates. This algorithm is illustrated in Figure 5.3. This optimization relies on the associative property of the updates such that updates can be accumulated first at each processor. For floating point updates, this will not get the same results due to aggregation. To preserve numerical correctness, one can choose to cache the updates without accumulation, then apply each update individually at the owner processor. The sequence of updates would match one of the possible ones in the original program without optimization.

Although the manual optimization eliminates the performance bottleneck, but it significantly lowers the readability of the code. In our case study for the heart code, the application programmer developed an entire mail box system to simulate message passing in order to express this optimization. For the rest of this chapter, we will present analyses and optimizations to the Titanium compiler to do this optimization automatically.

```
FluidCache cache = new FluidCache();
foreach (index in myParticleArray.domain()) {
    Particle p = myParticleArray[index];
    Point pos = [p.x, p.y, p.z];
    cache.add(pos+north, p.force);
    ...
    Cache.add(pos+east, p.force);
}

send cache to the processor that owns the underlying fluid

barrier;

increment fluid cell using values from incoming caches
```

Figure 5.3 Manually optimized spread force code using update by owner to avoid locking

5.3 To Automate the Caching Optimization

Two memory operations conflict if they can run concurrently and access the same memory location, and at least one of them is a write. A conflict edge exists between these two memory operations in such cases. Memory operations in different phases of a Titanium program cannot occur concurrently, because there are barriers between phases. Therefore, we only need to consider conflicting accesses within a phase.

We want to automate the kind of optimization described in Section 5.2. In this case, we would like to avoid using global locks, and have each processor to cache its own updates. Then the updates are sent to the processor that owns the memory location being modified, and the updates are processed sequentially.

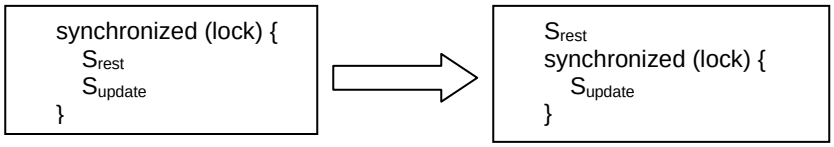
Here we present an analysis that determines when it is possible to perform this optimization. We first identify synchronized regions that do not contain other synchronization events inside. For each synchronized region, assume the region contains statements $S=\{s_1, s_2, \ldots, s_{total}\}$. Let S_{update} be the subset of statements in the synchronized region S. Then all statements in S_{update} must satisfy the conditions below:

1. s_i updates exactly one shared location X.
2. Location X is updated as a blind write.

3. s_i can be moved to the bottom of the synchronized region using code motion without violating def/use constraints.

Condition 1 and 2 allow us to accumulate updates if op is associative. If the operator is not associative, the updates can be cached, but the local processor cannot accumulate the updates. Condition 3 ensures that local dependencies are not violated, which is required by the Titanium memory model.

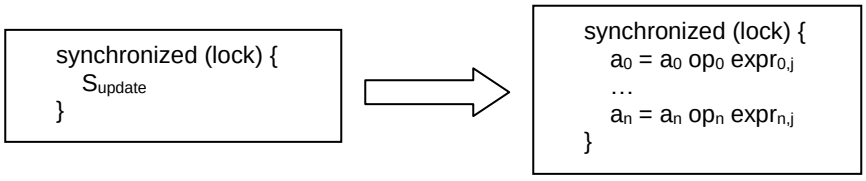
Let $S_{rest} = S_{total} - S_{update}$, we can put the statements in the order appearing in the left box below due to condition 2 above. We check if the following transformation is possible without losing atomicity.



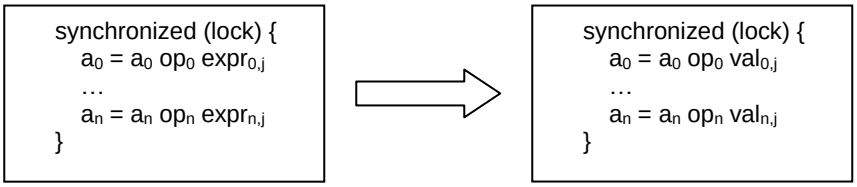
The check is done by detecting if there are new conflict edges introduced from moving S_{rest} outside of the synchronized region.

The next step is to show that the synchronization provided by the synchronized region on S_{update} can be replaced by an owner update. We remove the synchronized region, and rerun concurrency analysis [38, 39]. New conflict edges will be added to S_{update}. If the set of new conflict edges are all between s_i to itself for each s_i in S_{update}, then update by owner is sufficient to replace the synchronized region. Assume s_i updates memory location X_i, then owner update on memory location X_i would resolve the new conflict edge because the updates are done serially at the owner processor.

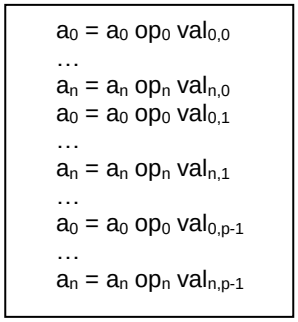
Let the a's be share variables, and $expr_{i,j}$ be the i_{th} expression evaluated on processor j.



We check previously that each s_i in S_{update} can be moved to the bottom of the synchronized region without violating def/use constraints. This means that every pair of s_i in S_{update} commutes, so that the statements in S_{update} can be executed in parallel, which makes the following transformation legal:



$eval(expr_{i,j})$ on processor j produces the value $val_{i,j}$. One possible execution stream among all p processors is the following:



This execution happens when the lock for the synchronized region is obtained in the order of processor 0, processor 1, …, processor p-1. The above is equivalent to the following:

$a_0 = a_0\ op_0\ val_{0,0}$
$a_0 = a_0\ op_0\ val_{0,1}$
…
$a_0 = a_0\ op_0\ val_{0,p-1}$
$a_1 = a_1\ op_1\ val_{1,0}$
$a_1 = a_1\ op_1\ val_{1,1}$
…
$a_1 = a_1\ op_1\ val_{1,p-1}$
.
.
.
$a_n = a_n\ op_n\ val_{n,0}$
$a_n = a_n\ op_n\ val_{n,1}$
…
$a_n = a_n\ op_n\ val_{n,p-1}$

They are equivalent because a_0, a_1, …, a_n are separate shared variables representing distinct memory locations. The val's can be evaluated in parallel. The value is then sent to the processor that owns the share variable, and the owner processor processes the values from different processors sequentially.

There are more opportunities to aggregate updates if we can cache updates from multiple executions of the same synchronized region. In Figure 5.2, the synchronized region is executed once per particle. We would like to find the outer most loop where we can send the updates after the loop terminates. Updates are sent to the processor that owns the memory locations being updated.

Aggregation of updates from multiple executions of the same synchronized region essentially delays the reads and writes from the update statements further down the execution path. Further analysis is required to ensure such code motion does not violate the Titanium memory model. For the candidate loop that encloses the synchronized region, we check for the following conditions:

1. Memory accesses outside of the synchronized region are of the types: local, non-shared, or read only within this phase of the program.

2. The loop does not contain other synchronization events besides the synchronized region we are trying to optimize.

Condition 1 also implies that there are no defs or uses of shared variables from the update statements outside the synchronized region in the loop. If the operator is associative, then

the updates for the same share variable can be accumulated locally at each processor. Otherwise, we must cache each individual update to preserve ordering.

5.4 Performance Model

Once the set of updates has been accumulated at a processor, there are several ways to package the updates and send them to the processor that owns the data to be updated. One obvious choice is to send the address along with the accumulated value for each update. But this method may not be optimal in all cases. Consider the case where the updates are to addresses that form a contiguous region in physical memory, we can optimize away most of the meta data by sending a bounding box descriptor including the starting address and the width of the bounding box. In general, there will be gaps between the updates, where multiple bounding boxes may be required. We develop a performance model for picking the optimal way to divide the updates into bounding boxes. It is fine to include gaps in a bounding box, because we can fill the gaps with the identity element for the given operation. For example, if the operation were addition, then we can fill the gaps with zeros.

As we scan the accumulated values in the hash table from low address to high address, we need to decide on how many bounding boxes and the start and end points for each bounding box. Intuitively, this is a tradeoff between the size of the meta data and the amount of wasted space and computation due to including gaps in the bounding box. We use a greedy algorithm to decide whether to include the gap in the current bounding box or not as we do the scanning.

If we include the gap into the current bounding box, then we have three extra costs due to this decision:

- Packing identity elements in place of the gap into the bounding box
- Communication cost in sending the gap elements
- When the updates arrive in the remote processor, the identity elements are read out of the buffer, then we process the updates to the data the remote processor owns due to the identity elements.

In contrast, if we decide to start a new bounding box starting at the other end of the gap, we incur the following costs:

- Packing bounding box descriptor into the buffer
- Communication cost in sending the descriptor for the newly created bounding box
- When the updates arrive in the remote processor, the bounding box descriptor is read out of the buffer.

Aside from the communication cost, both the packing and process update are local processor costs, which are dominated by memory access times. We use cache and memory latency numbers provided by the vendor in a performance model.

gapSize is the number of elements in the gap, descriptorSize is the size of the bounding box descriptor in words, b_w is the network bandwidth for a given size message, $L1_{line}$ and $L2_{line}$ are cache line sizes, and α_1, α_2 and α_{mem} are latencies for L1, L2, and memory, respectively. Cache line sizes are adjusted to match the word size in each formula.

Let $\alpha=\alpha_1+(1/L1_{line})*(\alpha_2-\alpha_1)+(1/L2_{line})*(\alpha_{mem}-\alpha_2)$, we can summarize the cost of including the gap into the current bounding box as $gapSize*(\alpha+1/b_w+2\alpha)$. The first α term comes from the packing of the identity elements into the buffer. $1/b_w$ is for the communication. The 2α is for the update processing, where one α is for reading the identity element out of the buffer and the other α is for updating the actual memory location. Similarly, the cost of introducing a new bounding box can be summarized as $descriptorSize*(\alpha+1/b_w+\alpha)$. The choice between including the gap in the current bounding box or introducing a new bounding box can be decided by the following:

```
if gapSize > descriptorSize*((α+1/bw+α)/(α+1/bw+2α)) then
   introduce a new bounding box
else
   include the gap into the current bounding box
```

Figure 5.4 Decision procedure for deciding between including the gap in the current bounding box or start a new bounding box at the end of the gap.

Our greedy algorithm makes decisions locally at each gap without knowledge of the entire gap distribution. The proof below shows that our greedy algorithm actually yields the optimal partitioning of updates with bounding boxes.

Claim: The greedy algorithm that makes a decision locally at each gap using the formula in Figure 5.6 yields globally optimal partitioning in terms of minimizing running time.

Proof: Assume that the greedy algorithm did not yield the optimal partitioning. Then the optimal partitioning can differ from the greedy algorithm produced partitioning in two ways:

1. There is a gap that is included inside of a bounding box in the optimal partitioning, but in the greedy algorithm partitioning, two separate bounding boxes are placed on the two ends of the gap. Due to the formula in Figure 5.6, it can be shown that the optimal partitioning can be improved by splitting the bounding box that includes the gap into two. It is a contradiction

that the optimal partitioning can be improved. Thus, by contradiction the greedy algorithm yields the optimal partitioning in this case.

2. There is a gap that is included inside of a bounding box in the greedy algorithm partitioning, but not in the optimal partitioning. Similar to 1, we can show that the optimal partitioning can be improved by joining together the two bounding boxes surrounding this gap in the optimal partitioning. So by contradiction, the greedy algorithm yields the optimal partitioning.

5.5 Implementation

After the analysis identifies the synchronized regions that can be optimized, the compiler generates code that uses update by owner to replace the use of global locking in the synchronized region.

The assignments in each of the update statements are replaced by a call to the runtime system to cache the updates. The caching is done using a hash table with the memory location as the key, and the update value as the value. The memory location is a global pointer, which contains the processor ID and pointer address. At compile time, the analysis determines whether the operator is associative. For associative operators, only the accumulated update is cached. Otherwise, each individual update needs to be stored separately to preserve numerical correctness.

The compiler also generates a send update call at the place as determined in Section 5.3. Updates that are destined for the same processor are combined together in the same active message request. The remote processor does the updates inside the active message handler. The remote processor can only service one request at a time, and it does not block inside the handler. This gives us the sequential update property.

During one call to send update, the calling processor may have updates for multiple remote processors. In this case, the sending of requests to different processors is overlapped.

In summary, the optimization gives us three benefits:

1. Global locking is avoided. Thus we can achieve mostly concurrent execution of the instructions inside the synchronized region.
2. Aggregation by caching reduces the number of communication events. In the case of an associative operator, the accumulated result is sent to the processor that owns the shared variable. This achieves a reduction in the volume of data being communicated. Otherwise, the individual updates are sent in one active message request to the remote processor.

3. Communication overlap is achieved by sending non-blocking update requests to multiple remote processors.

5.6 Experimental Results

In this section, we show the experimental results on running the synchronized region optimization on three Titanium benchmarks, the spread force operation in the heart simulation application, histogram, and particle gravitation.

The experiment was performed on a cluster of Intel Xeon processors connected by a high-speed Infiniband network. Each node contains two Xeon dual core processors running at a clock speed of 2.66 GHz. Various processor configurations were used in our experiment from 2 cores to 128 cores.

The input data for the spread force benchmark is from a sphere immersed at the center of a 256^3 fluid grid. The sphere is made up of 1.28 million fiber points. The fiber points were partitioned offline by a graph partitioner. The graph partitioner considers the tradeoff between assigning fiber points that are close together to the same processor and load balance. The underlying fluid grid is partitioned in slabs. Each fiber point spreads its force to its surrounding fluid cells.

In the histogram benchmark, the samples are partitioned among the processors. Each processor owns the counts for a subset of the histogram. As the samples come in, the corresponding count would be updated depending on the value of the sample. Values in the samples are randomly distributed uniformly between the range of the histogram.

In the particle gravitation benchmark, the particles are distributed evenly among the processors. Each particle exerts a gravitation force to all other particles. On each iteration, each processor goes through the particles it owns, and updates all other particles due to gravitational force.

The Titanium compiler was able to automatically apply the synchronized region optimization to all three benchmarks. The operator in the both spread force and particle gravitation benchmarks is addition on floating point numbers. The scientist who developed the benchmarks concluded that it would not affect numerical stability by assuming associativity. The operator in the histogram benchmark is addition on integers, which is associative.

Figure 5.5, 5.6, and 5.7 show the speedups of our synchronized region optimized code and the same code compiled without the optimization relative to the sequential code for the three benchmarks, from 2 cores to 128 cores. Without the synchronized region optimization, the histogram and particle gravitation benchmarks fail to recover the sequential performance even

at 128 cores. For the spread force benchmark, the speedup over the sequential code is 6.74 for the code without synchronized region optimization on 128 cores. Clearly, a straightforward compilation of PGAS code with irregular updates does not yield reasonable performance. The number of remote updates increases with additional cores, as each node only has four cores. Without the synchronized region optimization, the total number of remote messages between all cores increases linearly according to the formulas below:

Histogram: sampleTotal*(p-1)/p
Particle gravitation: particleNum^2*(p-1)/p
Spread Force: particleNum*(p-1)/p when the particles are dispersed randomly among the fluid grid

With the synchronized region optimization, updates are cached locally and then combined before sending them to the remote processor. The number of remote messages is much less than the version compiled without the optimization. We have at most two messages between each core pair assuming each core has some updates for every other core. Since the input size such as the number of samples or the number of particles is much larger than the number of cores, we see several orders of magnitude reduction in message count due to the synchronized region optimization. This results in significant speedups in all three benchmarks.

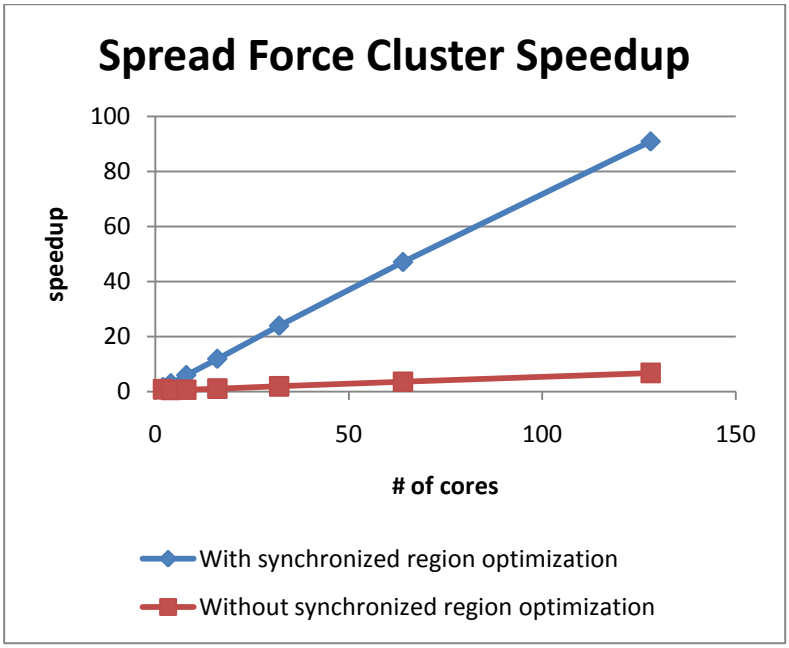


Figure 5.5 Speedup comparison for the spread force benchmark. Experiments were run on various processor configurations from 2 cores to 128 cores. At each processor configuration, the speedup is computed relative to the sequential implementation of the benchmark.

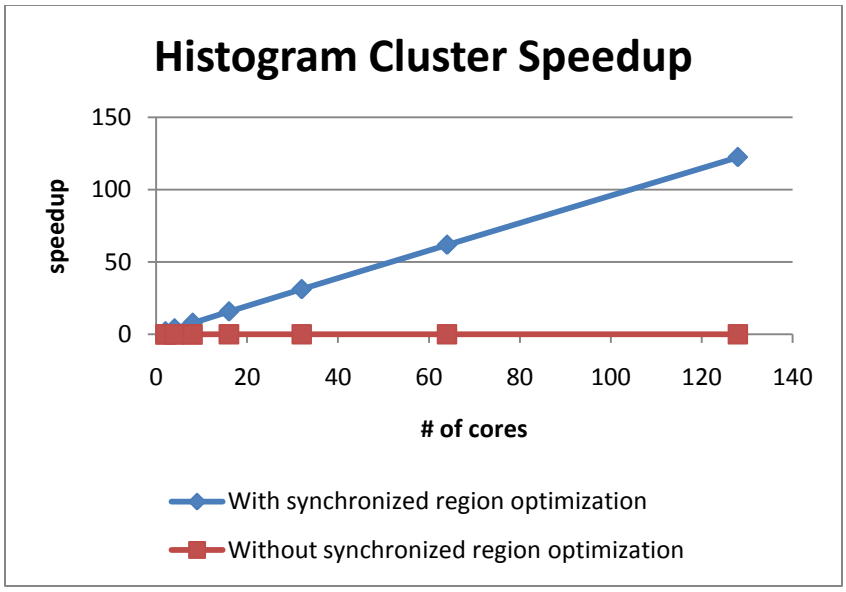


Figure 5.6 Speedup comparison for the histogram benchmark. Experiments were run on various processor configurations from 2 cores to 128 cores. At each processor configuration, the speedup is computed relative to the sequential implementation of the benchmark.

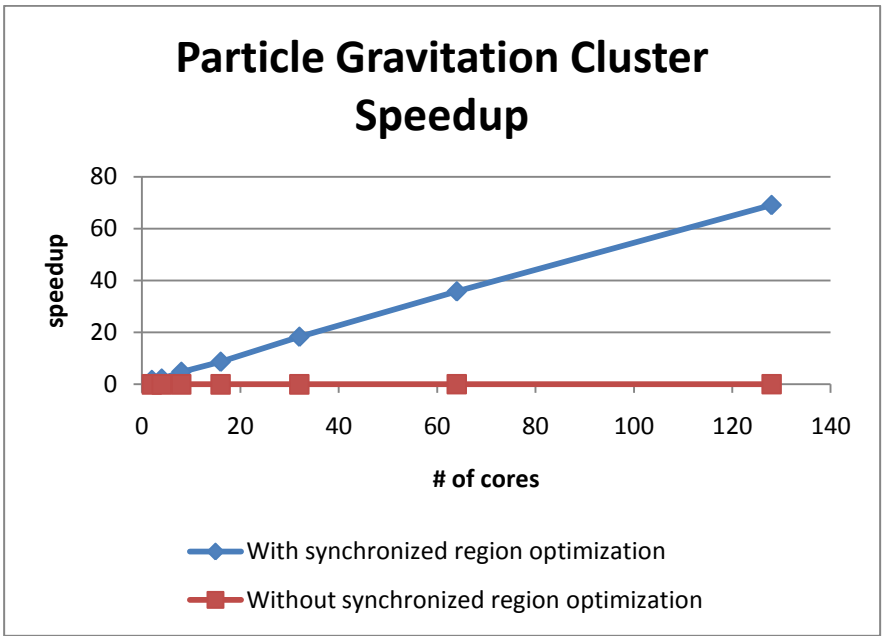


Figure 5.7 Speedup comparison for the particle gravitation benchmark. Experiments were run on various processor configurations from 2 cores to 128 cores. At each processor configuration, the speedup is computed relative to the sequential implementation of the benchmark.

5.7 Summary

In this chapter, we have described an automatic transformation of programs with synchronized regions in Titanium to take advantage of optimization to replace global locking with update by owner when possible. This allows application programmers to write Titanium code in a straightforward way and improves performance by orders of magnitude over the same code compiled without this optimization. In particular, for the three benchmarks: spread force operation in the heart simulation application, histogram, and particle gravitation, the optimization yielded speedups up to 2526x over the same code compiled without the optimization on a cluster of Xeon processors.

Chapter 6

Multicore Optimizations for Irregular Writes

6.1 Introduction

We are now in the era of the multicore revolution, which is witnessing a rapid evolution of architectural designs due to power constraints and correspondingly limited microprocessor clock speeds. Understanding how to efficiently utilize these systems in the context of demanding numerical algorithms is an urgent challenge to meet the ever growing computational needs of high-end computing.

In this chapter, we develop an adaptive implementation strategy that maximizes the number of local updates while minimizing the reduction cost. We demonstrate through the histogram and heart spread force kernels that the adaptive implementation gives significantly higher performance than pure locking and replicate and reduce implementations. We also explore various performance tradeoffs in asynchronous parallel update codes on modern multicore processor architectures. On modern multicore architectures, shared memory is accessible by all threads, which allows the programmer to write parallel programs without using message passing. This is a significant gain in programmability and productivity for the programmer, but the use of shared memory does not alleviate the need for performance tuning entirely. Access from different cores to the shared memory yields different performance due to NUMA issues. In this chapter, we explore various implementations that perform tradeoffs between per update cost and overhead cost.

The optimizations described in this chapter were directly done on C code. These are lower level optimizations that are done on C code of the kind generated by the Titanium compiler, rather than being in the Titanium language itself. These are optimizations that would require a similar analysis to chapter 5, but are done as a standalone code generator.

6.2 Experimental Setup

In this section, we describe in detail the machines used in this chapter, the common benchmarking methodology for the two benchmarks.

6.2.1 Machines

We select three leading multicore designs to explore the benefits of our threaded implementations of the kernels across a variety of architectural paradigms.

A summary of their architectural parameters is provided in Table 6.1.

Core Architecture	**AMD Barcelona**	**Intel Nehalem**	**Sun Niagara2**
Type	superscalar out of order	superscalar out of order	HW multithreaded dual issue
Clock (GHz)	2.30	2.66	1.16
Double-precision GFlop/s	9.2	10.7	1.16
L1 Data Cache	64 KB	32 KB	8 KB
private L2 cache	512 KB	256 KB	—
System	**Opteron 2356 (Barcelona)**	**Xeon X5550 (Gainestown)**	**UltraSparc T5140 (Victoria Falls)**
# Sockets	2	2	2
Cores(Threads) per Socket	4(4)	4(8)	8(64)
Primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading
Shared last-level cache	2×2 MB (shared by 4 cores)	2×8 MB (shared by 4 cores)	2×4 MB (shared by 8 cores)
DRAM Capacity	16 GB	12 GB	32 GB
DRAM Pin Bandwidth (GB/s)	21.33	51.2	42.66(read) 21.33(write)
Double-precision GFlop/s	73.6	85.3	18.7

Table 6.1: Architectural details of parallel platforms

6.2.1.1 Opteron

The Opteron 2356 (Barcelona) is AMD's quad-core processor offering. Each Opteron core runs at 2.3 GHz, has a 64 KB L1 cache, and a 512 KB L2 victim cache. In addition, each chip instantiates a 2MB L3 quasi-victim cache that is shared among all four cores. Each Opteron socket includes two DDR2-667 memory controllers providing up to 10.66 GB/s of raw DRAM

bandwidth. Sockets are connected via a cache-coherent HT link creating a coherency and NUMA network for this 2 socket (8 core) machine.

6.2.1.2 Nehalem

The recently released Nehalem is the latest enhancement to the Intel ``Core'' architecture, and represents a dramatic departure from Intel's previous multiprocessor designs. It abandons the front-side bus (FSB) in favor of on-chip memory controllers. The resultant QuickPath Interconnect (QPI) inter-chip network is similar to AMD's HyperTransport (HT), and it provides access to remote memory controllers and I/O devices, while also maintaining cache coherency. Nehalem is novel in two other aspects: support for two-way simultaneous multithreading (SMT) and TurboMode. The latter allows one core to operate faster than the nominal clock rate under certain workloads. On our machine, TurboMode is disabled due to its inconsistent timing behavior.

The system used in this study is a dual-socket, quad-core 2.66 GHz Xeon X5550 with a total of 16 hardware thread contexts. Each core has a private 32 KB L1 and a 256 KB L2 cache, and each socket instantiates a shared 8 MB L3 cache. Additionally, each socket integrates three DDR3 memory controllers operating at 1066 MHz, providing up to 25.6 GB/s of DRAM bandwidth to each socket. In comparison to the Barcelona system used in this chapter, Nehalem has a similar floating-point peak rate but a significantly higher memory bandwidth and cache.

6.2.1.3 Victoria Falls

The Sun ``UltraSparc T2 Plus'', a dual-socket x 8-core SMP referred to as Victoria Falls, presents an interesting departure from mainstream multicore processor design. Rather than depending on four-way superscalar execution, each of the 16 strictly in-order cores supports two groups of four hardware thread contexts (referred to as Chip MultiThreading or CMT) --- providing a total of 64 simultaneous hardware threads per socket. Each core may issue up to one instruction per thread group assuming there is no resource conflict. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multithreading. Victoria Falls has no hardware prefetching, and software prefetching only places data in the L2 cache.

Multithreading may hide instruction and cache latency, but may not fully hide DRAM latency. Our machine is clocked at 1.16 GHz, does not implement SIMD, but has an aggregate 64 GB/s of DRAM bandwidth in the usual 2:1 read:write ratio associated with FBDIMM. As such, it has significantly more memory bandwidth than either Barcelona or Nehalem, but has less than a

quarter the peak flop rate. With 128 hardware thread contexts, this Victoria Falls system poses parallelization challenges that we do not encounter in the Gainestown and Barcelona systems.

6.2.2 Threading Model

To evaluate performance of the threaded implementations, we employ a SPMD-inspired threading model in which communication of particles or updates to grid values is handled through shared memory rather than message passing. Typically, we create N threads, partition the particles or data points, allow them to initialize their data, and then benchmark repeated spread force and histogram building. The SPMD threading model is the same as the one in Titanium. The partitioned global address space provided by Titanium gives the programmer the abstraction of shared memory. For Titanium's cluster backends, the actual memory partitions are spread among multiple nodes on a cluster. In our multicore experiments, actual shared memory is used between cores on the same machine. This significantly reduces the latency in memory access in comparison to the cluster case, but there is still a performance difference between intra-socket memory access and inter-socket memory access.

6.3 Optimizations: Problem Decomposition

Histogram problem decomposition is relatively straightforward. The parallelism comes from the concurrent processing of the data points. The data points are typically partitioned in equal chunks to the threads. The number of buckets in the histogram is typically small relative to the number of data points.

PIC applications typically have two types of decompositions: particle decomposition and grid decomposition. The particle decomposition determines the amount of work each thread performs during the update phase. The grid decomposition divides the underlying grid geometrically among the threads. In this chapter, our target architectures are modern multicore processors, where each thread can access every grid cell through shared memory. But due to NUMA issues, there is a performance distinction between a thread accessing a grid cell that it owns and a grid cell on another socket. The performance difference is not as significant as in the cluster case, where accessing local memory can be orders of magnitude faster than accessing remote memory, but it is significant enough to be a performance tuning option in the multicore case.

We want to partition the grid in such a way to maximize the number of particle to grid updates that can be performed locally, while maintaining load balance by having nearly equal number of particles for each thread to work on. We would also like to minimize the memory footprint of

the problem, because in full applications the memory footprint will limit the problem size. These three factors locality, load balance, and memory use tradeoff against one another, and result in different optimal solutions for different inputs. One can have perfect load balance and have local updates for all particles by replicating the entire grid on each thread, and combining the grids with a reduction at the end. This would significantly increase the memory footprint and the running time due to the cost of reduction if the replica is large. This may be an acceptable solution for problems with small grids, such as in the histogram case. But it would require excessive amount of memory in the case for the heart code. On the other extreme, one can have perfect load balance and smallest memory footprint by having all the threads to update a shared grid with locking. Depending on the particle distribution, this may degrade in performance due to contention. In this chapter, we explore many dimensions in this optimization space.

6.3.1 Histogram

The input data array is partitioned into the same number of chunks as the number of threads. If the size of the data array is divisible by the number of threads, each thread gets the same number of data elements. Otherwise, some threads may have one more element than others. The partitioning scheme for the buckets depends on the synchronization strategy. The different partitioning schemes differ in memory requirements, ranging from each thread having an entire copy of the histogram to sharing one global histogram among the threads. The different synchronization strategies will be discusses in Section 6.4.

6.3.2 Heart Spread Force Computation

In the Heart Simulation, the fluid grid is partitioned into rectangular slabs along the X dimension. Each thread gets one fluid slab. The size of the fluid slab may vary depending on the synchronization strategy. Where there are more threads than the number of y-z planes, each rectangular slab is further partitioned in the Y dimension to accommodate the extra threads. The fiber points are partitioned into equal chunks to each thread. We would like to maximize the number of fiber points that live within the fluid slab owned by the same thread. This would minimize the memory traffic across sockets in a multicore processor. Further discussions about this will be in the synchronization optimization section.

6.4 Optimizations: Synchronization

During random updates, the target of the update may receive multiple updates simultaneously. Synchronization is required to avoid race conditions. For histogram, a bucket may receive simultaneous updates when multiple threads are processing data points that fall to the same bucket. In the particle update phase in particle mesh applications, a particle spreads its updates to its nearest neighboring grid cells. For two particles that may update its neighboring cells concurrently, synchronization is required on the intersection of the cells that need to be updated by the two particles. This type of synchronization can be achieved in many different ways. Using locks is a natural choice for shared memory programming. Partial or full replication of the grid by each thread is another approach. Replication is often the ideal choice in the cluster setting, due to the large performance gap between accessing local memory versus remote memory. In this section, we explore different synchronization strategies including hybrid locking with replication.

6.4.1 Histogram

In histogram, each thread has a set of data points to work on. Two different threads may have data points that require updates to the same bucket. Synchronization is required to prevent data races to the buckets. Three types of synchronization methods were used in our experiments: locking using a shared bucket array, replication of the bucket array for each thread, and a hybrid method using locking on parts of the array and replication on other parts of the array. The distribution of the data points and the size of the bucket array determine the best synchronization method.

6.4.2 Heart Spread Force Computation

In the heart code, the density of the fiber points is not uniform throughout the fluid grid, and the fiber points are sparse. These differences require us to develop two synchronization techniques to better performance. Due to the non-uniform fiber point density, the strategy for each thread to have a single mirror of its fluid slab does not yield optimal performance. The algorithm must adjust the mirror size for regions of the fluid cell with different fiber point densities. A single mirror per thread would result in too many updates to fluid cells that were unmodified in the reduction phase. Having multiple boxes per mirror alleviates this problem, since boxes that have zero updates would not be used in the reduction phase. In Figure 6.1, it shows the decomposition of a 2D fluid slab into 2x2 boxes. The actual grid we used in our

experiments is 3D. Boxes are allocated when there is an update for the box. With multiple boxes, an additional data structure is needed to keep track of allocated boxes.

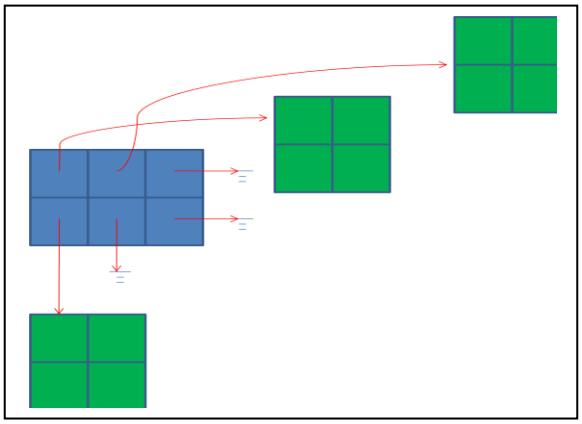


Figure 6.1: Dynamic allocation of replica boxes. In some particle mesh codes, particle density is not uniform throughout the grid, such as in the heart code. Having a flat replica may incur additional reduction cost due to grid cells with zero updates.

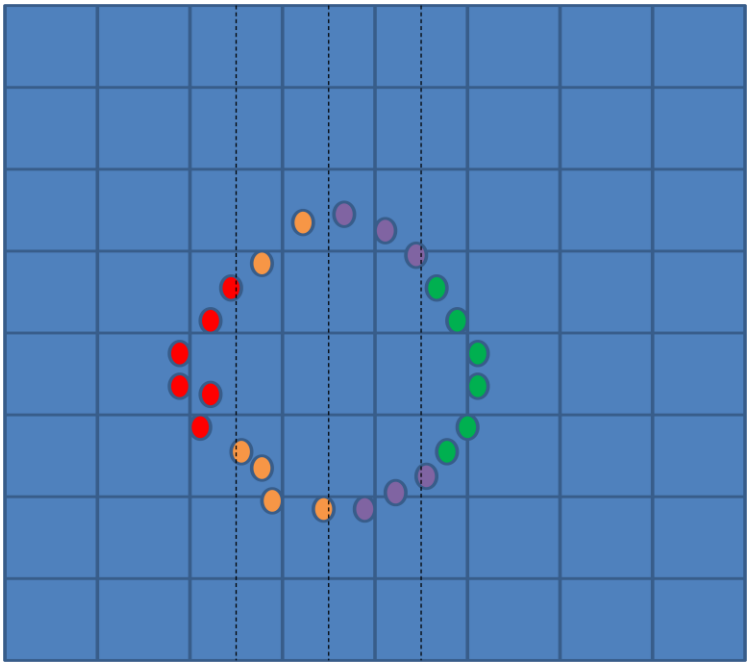


Figure 6.2: Particle partitioning in 2D with sorting. Particles assemble a circle in a 2D grid. Particles are sorted by their X coordinates. Each thread gets a near equal chunk of particles. Using the minimum and maximum values of the assigned particles' X coordinates, the grid is partitioned along the X dimension. This maximizes the number of particles that have updates to the local grid.

Due to NUMA on the Barcelona and Nehalem processors, accesses to off socket memory and on socket memory have significantly different performances. To minimize the number of off socket memory accesses, we first sort the fiber points by the dimension that the fluid grid is partitioned on. Then we partition the fiber points and fluid slabs together, so that we maximize the number of fiber points that live in the fluid slab owned by the same thread. Figure 6.2 illustrates an example of this partitioning strategy on a circle of fiber points in 2D. Since the density of the fiber points is not uniform throughout the fluid grid, this partitioning strategy would result in unequal sized fluid slabs for the threads. In the spread force step of the heart simulation, the workload for each thread is proportional to the number of fiber points instead of the fluid slab size. Unequal sized fluid slabs do not present a load balancing problem in spread force.

6.5 Low-Level Tuning

In order to achieve repeatable performance, thread pinning is used on all three architectures. Thread pinning binds a thread to an execution unit during the execution of the program. For the Pthreads implementations, we enumerate hardware thread contexts so that when ramping up the number of threads, we exploit multithreading within a core, then multicore on chip, and finally multiple sockets on the SMP.

6.6 Performance Analysis

In this section, we provide performance analysis on the various experiments for the two benchmarks on Nehalem, Barcelona, and Victoria Falls. Performance for histogram is measured in updates per second, and heart spread force performance is measured in GFLOPS.

6.6.1 Histogram Performance

For the histogram benchmark, we use index distributions from Web page spam detection and Web page language encoding. In recent years, statistical techniques have been developed to combat Web spam [27]. Web spamming refers to actions intended to mislead search engines into ranking some pages higher than they deserve. Web pages can be characterized by many attributes, such as number of outlinks and inlinks, number of words on a page, and the number

of punctuations among many others. The main insight is that the outliers in the distribution of those attributes are likely to be spam. Histograms are widely used in this technique to spot outliers. We measure the performance of the three parallel implementations of histogram in terms of updates per second.

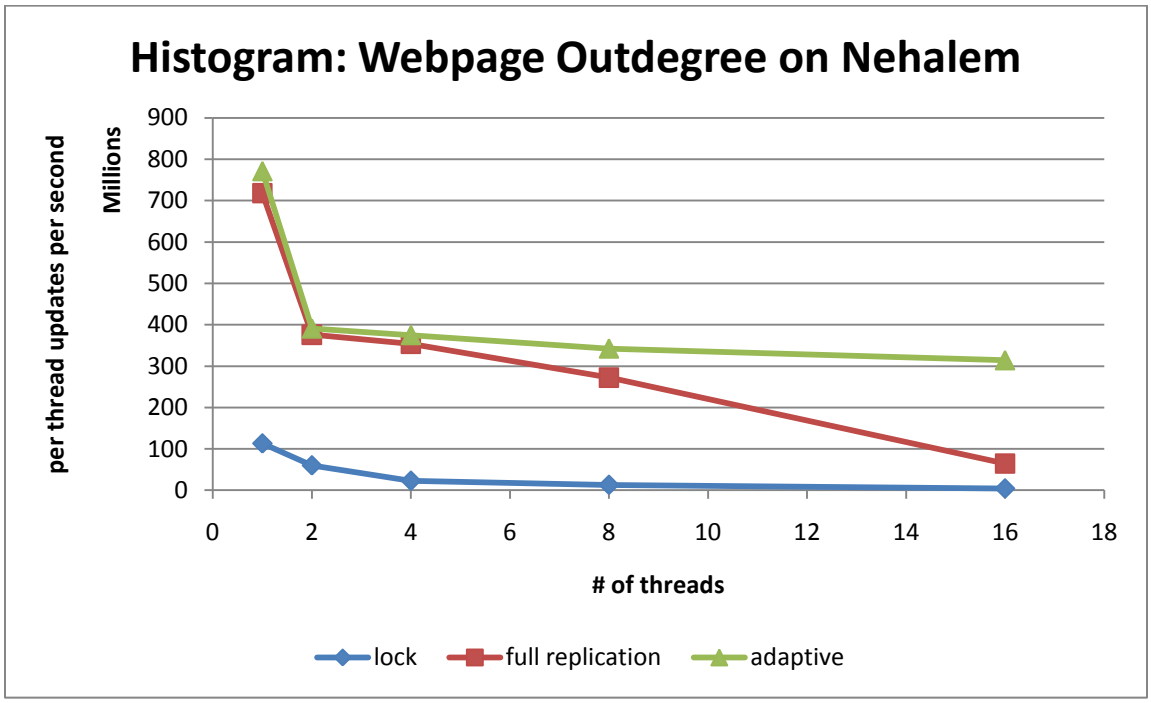


Figure 6.3: Histogram per thread performance on Nehalem for the Web page out-degree distribution. Performance is measured in terms of updates per second.

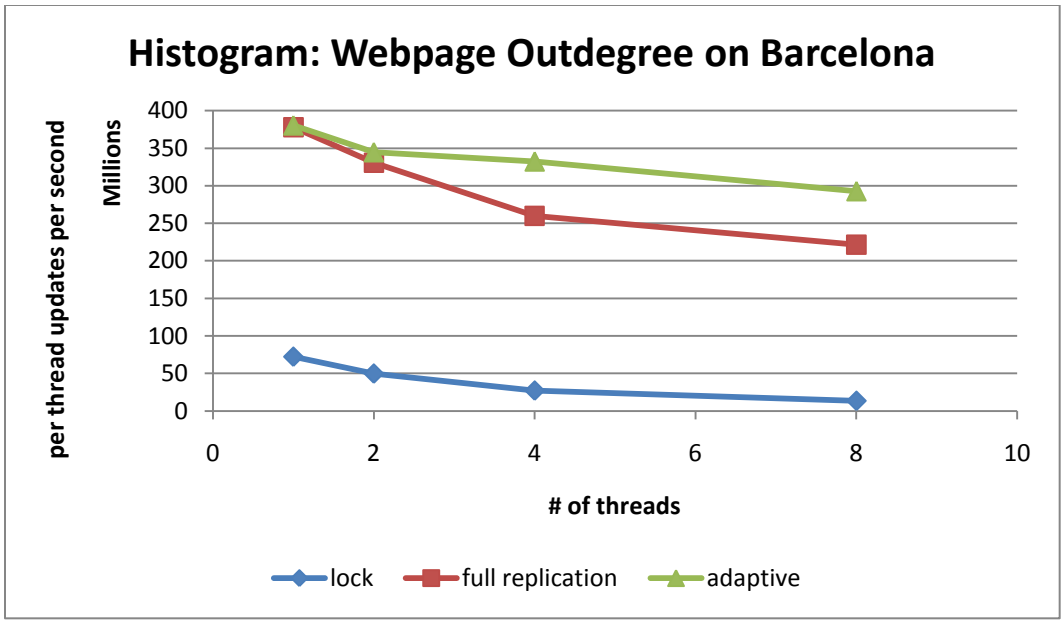


Figure 6.4: Histogram per thread performance on Barcelona for the Web page out-degree distribution. Performance is measured in terms of updates per second.

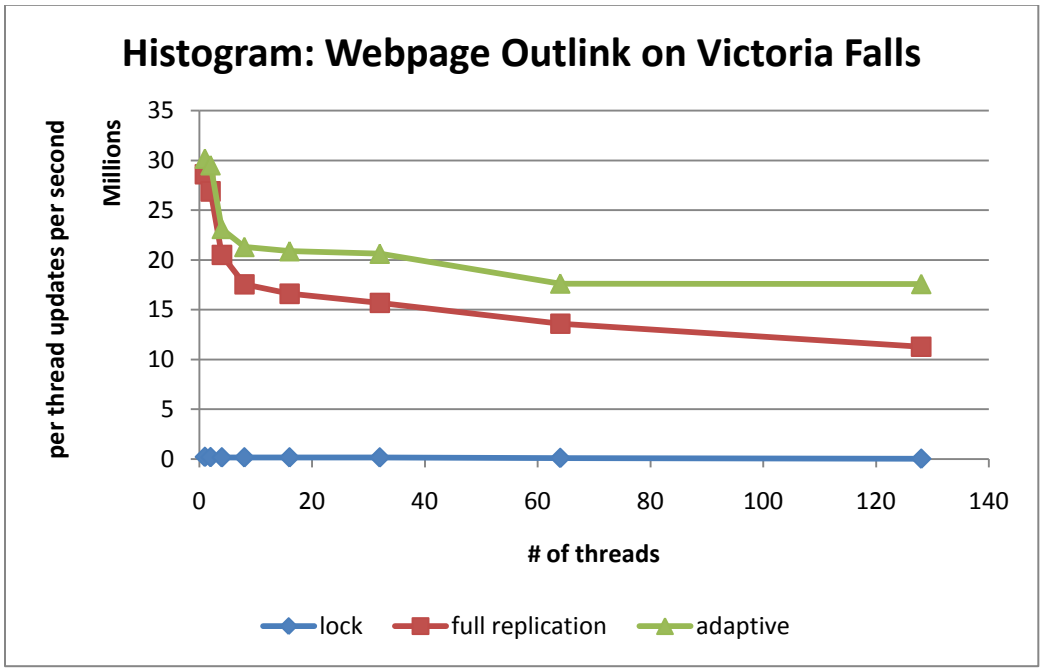


Figure 6.5: Histogram per thread performance on Victoria Falls for the Web page out-degree distribution. Performance is measured in terms of updates per second.

The index distribution is from the number of outlinks from 16 million Web pages. Each bucket of the histogram represents the number of pages containing a particular number of outlinks. Most human written Web pages have fewer than 100 outlinks. This causes high contention at buckets representing outlink counts fewer than 100. There are outliers due to machine generated pages, directory pages, and spam link farms. For example, pages in a spam link farm likely have the same number of outlinks, since the pages in the link farm form a clique to maximize search engine ranking. This results in a few outliers with high outlink counts up to 40000.

Figure 6.3, 6.4, and 6.5 show the per thread updates per second numbers for the different implementations on Nehalem, Barcelona, and Victoria Falls, respectively. On the Nehalem system, we first exploit SMT parallelism within a core before utilizing a second core. The number of updates per second drops by a factor of two going from one thread to two threads on Nehalem, suggesting two-way SMT yields little benefit for this histogram problem. SMT parallelization is beneficial for memory latency limited problems, but locking prevents parallel progress. For the Web page outlink distribution, there is lock contention at buckets for 100 outlinks and fewer. This severely limits the performance of the locking implementation. At low thread counts, the full replicate and reduce implementation performs nearly as well as the adaptive implementation. But the performance difference is evident when we scale to more threads on all three architectures. In full replicate and reduce, reduction is performed for every bucket, even the ones without any updates. Both the memory use and reduction cost increase with more threads. In contrast, the adaptive implementation uses replicate and reduce for buckets representing 100 outlinks and fewer, and uses locking for buckets representing outlinks greater than 100. This effectively avoids the reduction on buckets with zero updates, and avoids contention on other parts of the histogram.

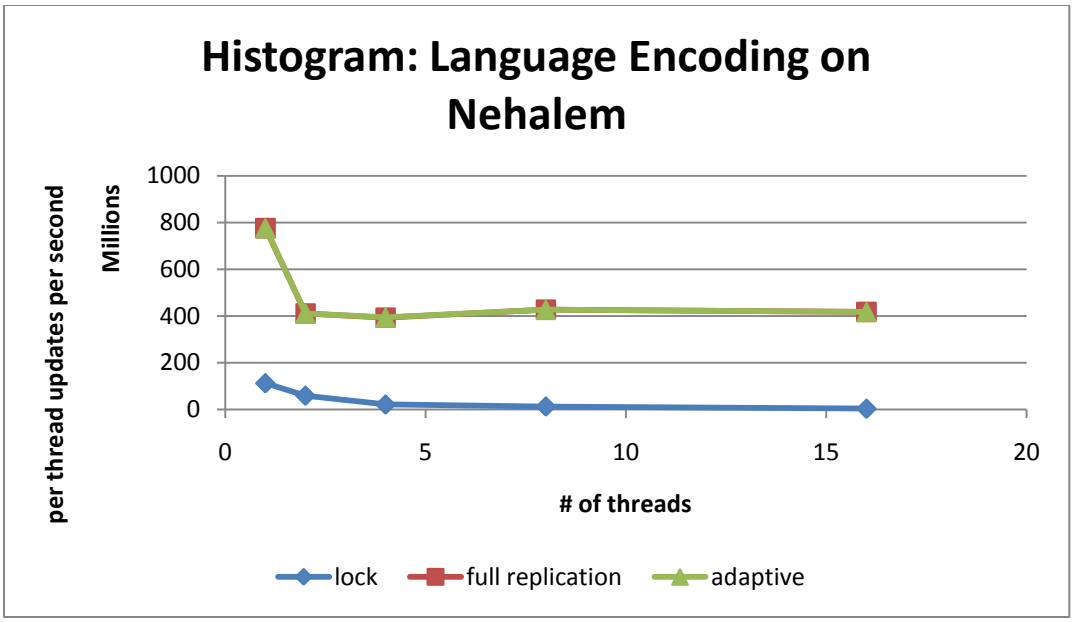


Figure 6.6: Histogram per thread performance on Nehalem for the language encoding distribution. Performance is measured in terms of updates per second.

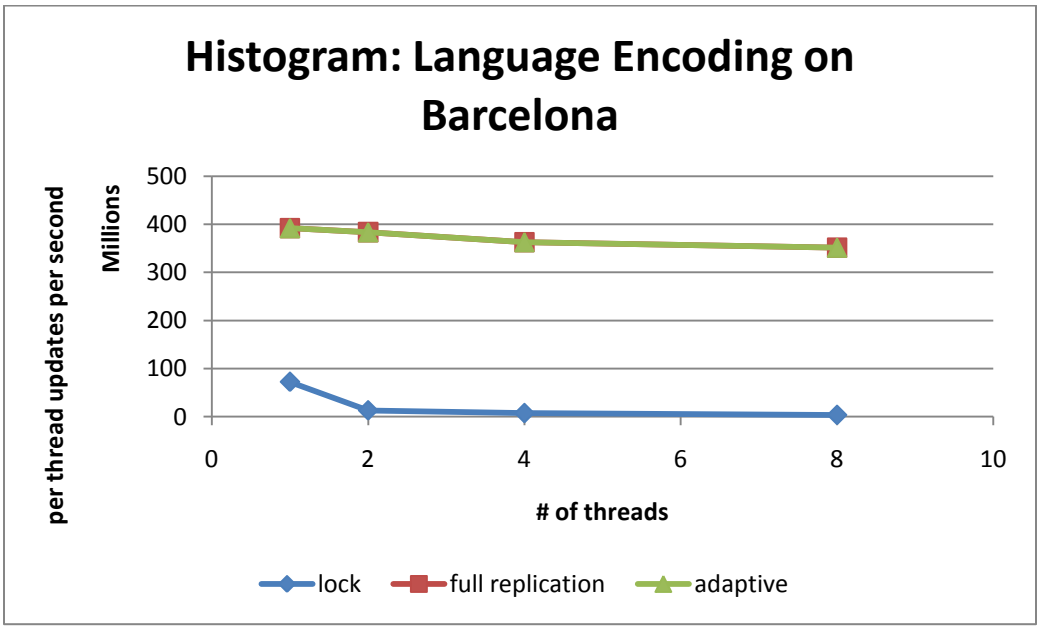


Figure 6.7: Histogram per thread performance on Barcelona for the language encoding distribution. Performance is measured in terms of updates per second.

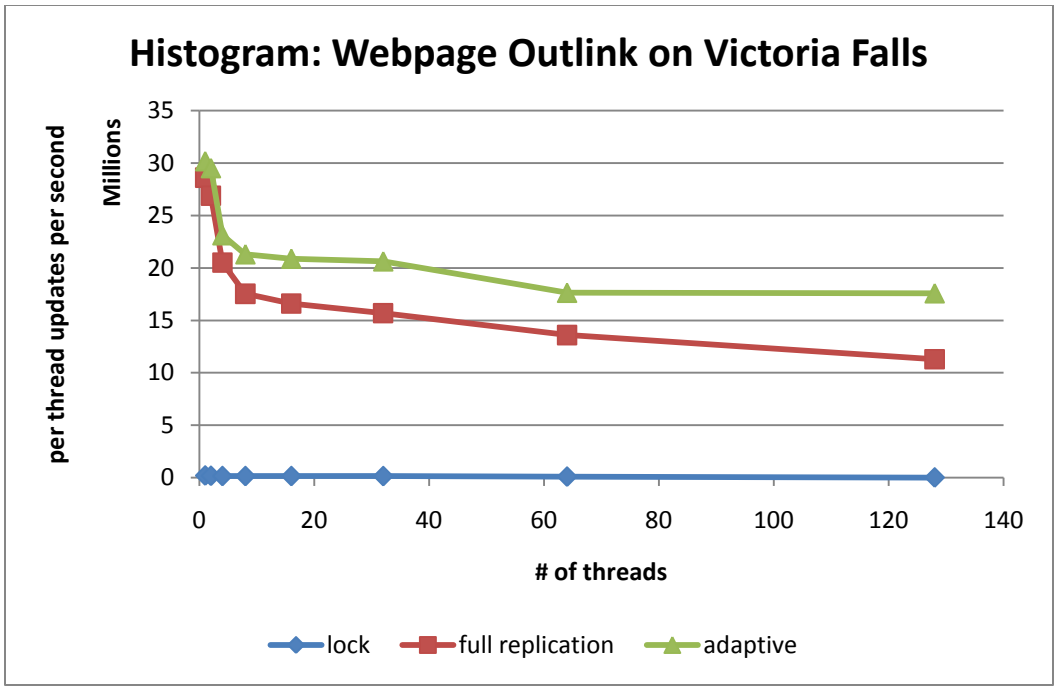


Figure 6.8: Histogram per thread performance on Victoria Falls for the language encoding distribution. Performance is measured in terms of updates per second.

We use the same set of Web pages for the language encoding distribution. There are close to 100 language encodings in this set of pages. The distribution is uniformly random over the language encodings. The adaptive implementation is near identical to the replicate and reduce implementation. The two implementations exhibit very similar performance, scaling near linearly for fully threaded cores on all three architectures. Due to the large data points to buckets ratio and high contention, the locking implementation does not perform well in this configuration.

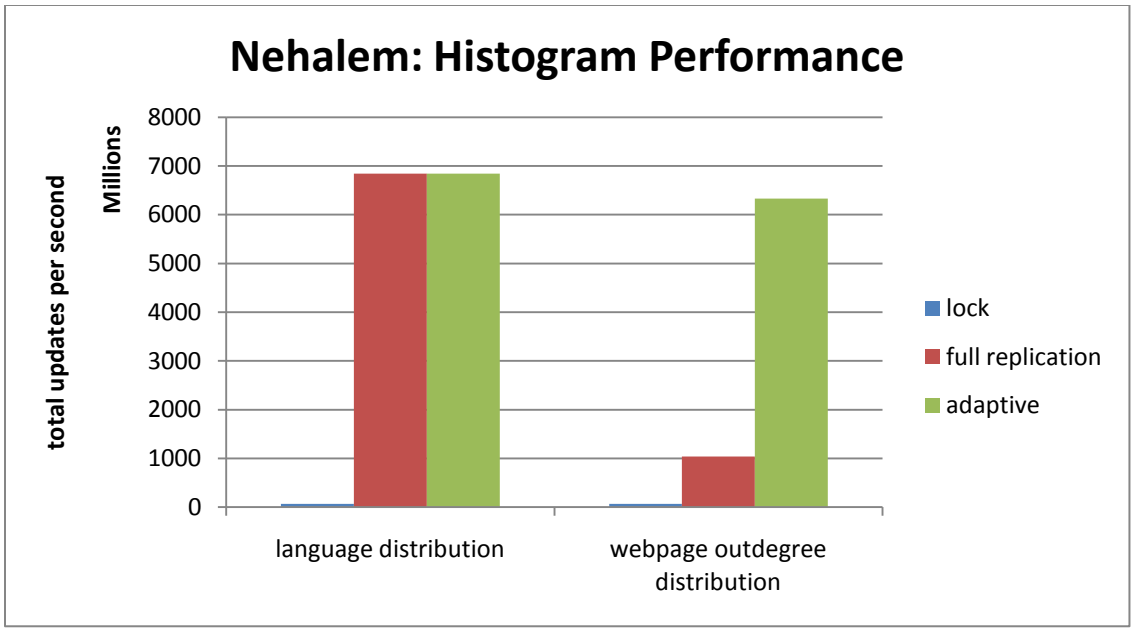


Figure 6.9: Histogram performance summary on Nehalem for 16 threads. Performance is measured in terms of updates per second.

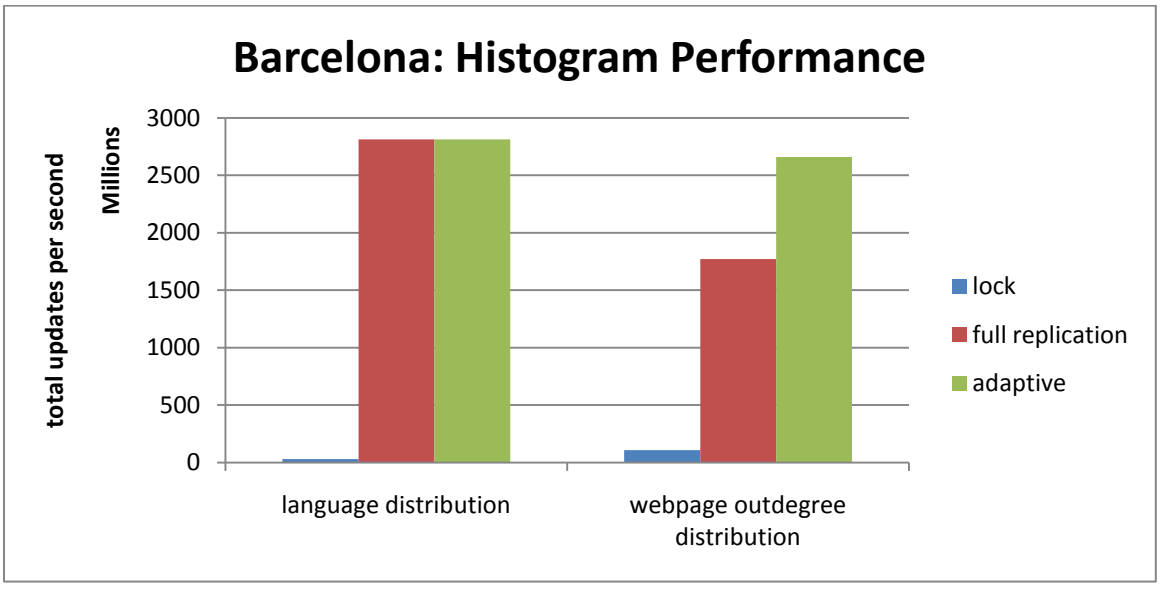


Figure 6.10: Histogram performance summary on Barcelona for 8 threads. Performance is measured in terms of updates per second.

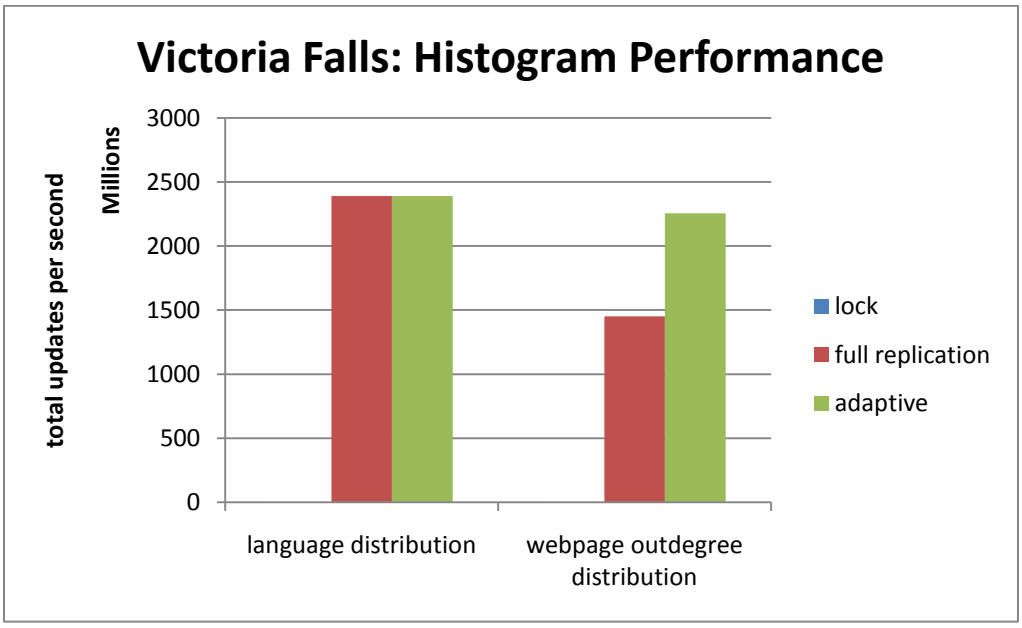


Figure 6.11: Histogram performance summary on Victoria Falls for 128 threads. Performance is measured in terms of updates per second.

Figure 6.9, 6.10, and 6.11 summarize the performance of the histogram kernel on the three architectures. It uses the best performance numbers for each implementation using the maximum number of thread contexts available. The adaptive implementation matches the performance of full replicate and reduce on the language encoding distribution, and outperforms both the replicate and reduce and locking implementations on the Web page outdegree distribution.

The memory footprint size difference between the three histogram implementations is insignificant. The number of data points is significantly larger than the number of buckets in the histogram in both cases. The total memory footprint size for all three implementations is nearly the same, due to the memory size of the data points being the dominant factor.

6.6.2 Heart Spread Force Performance

Memory utilization varies dramatically among the different heart spread force implementations, as illustrated in Figure 6.12, 6.13, and 6.14. For the flat replication and reduce implementation, each thread has a full replica of the grid array. For the 128 threads configuration on Victoria Falls, the total size of the replicas is over 16 GB. As more cores become available on future multicore architectures, the flat replication and reduce strategy is not going to scale due to memory constraints. It is not likely that future multicore systems' aggregate memory capacity and bandwidth will scale linearly with the number of cores. The replication with mirror approach always has a memory footprint size that is twice the size of the grid array. Each thread gets an equal size slab that is the mirror of a portion of the grid array.

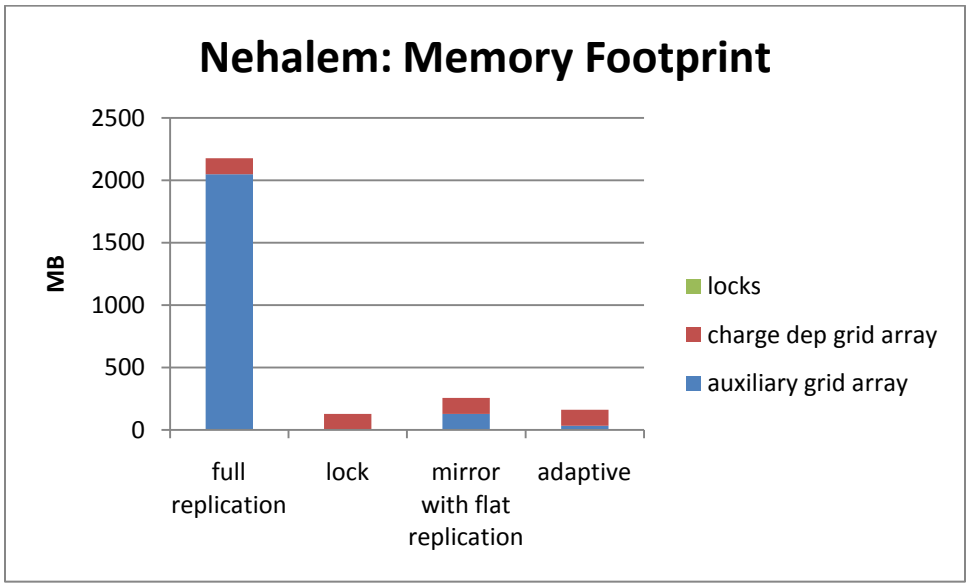


Figure 6.12: Memory footprint of different heart spread force implementations on Nehalem using 16 threads.

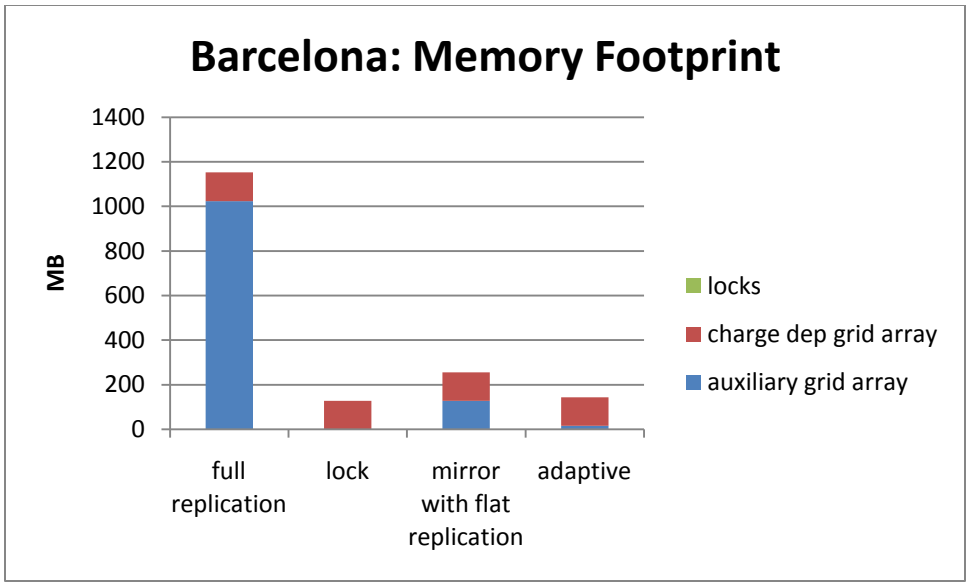


Figure 6.13: Memory footprint of different heart spread force implementations on Barcelona using 8 threads.

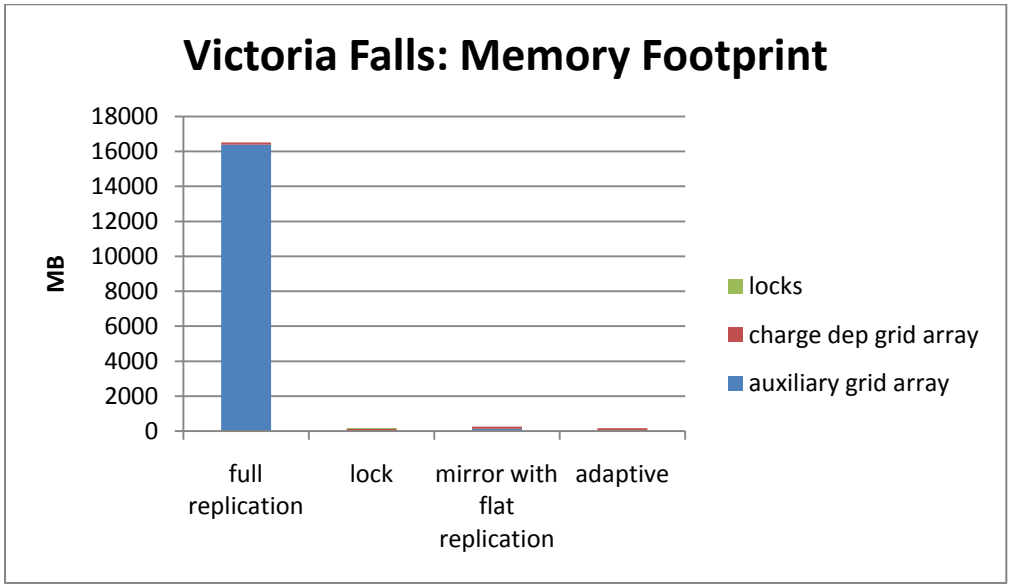


Figure 6.14: Memory footprint of different heart spread force implementations on Victoria Falls using 128 threads.

Memory footprint size is the most economical for the locking approach. It does not require replication arrays. For Nehalem and Barcelona, inline assembly is used to update the shared grid array atomically, so no additional memory is needed for the lock data structure. On Victoria Falls, there is a lock data structure for the grid array, costing 4 bytes per grid cell.

The adaptive approach requires replicas for ghost regions, as updates to the thread's own part of the grid goes directly on the shared grid. The size of the ghost region would depend on how many particles live outside the thread's grid region. Sorting the particles along the dimension that the grid is partitioned in is necessary for greatly reducing the number of particles living outside of a thread's region. Sorting does not add additional memory requirement to the different implementations, as we use an in place parallel quick sort.

Sorting by itself does not significantly reduce the number of particles that are outside of a thread's region for the Heart code. The particle density in the Heart code is not uniform. Particles are concentrated in a certain area of the grid. In the mirror replication approach, each thread gets an equal size slab as its region. Threads that get the slabs toward the ends will not have any particles in its region, since the particles are concentrated in the middle of the grid. As evident in the GFLOPS comparison between the sorted and unsorted implementations for mirror replication, sorting alone does not improve performance in the case for mirror replication. For the locking implementation, sorting improves performance for higher thread counts due to decreased contention and less random memory access since the particles are processed in the sorted order.

To maximize the number of particles that are local to a thread's region, the adaptive implementation adjusts the partitioning of the grid according to the sorted order of the particles. Each thread gets a near equal size chunk of the particles. Then using the minimum and maximum coordinates for the X dimension of the particles, the grid is partitioned among the threads. Due to the non-uniform particle density in the Heart code, the slab partitions are not equal size. This does not affect the load balance in the spread force phase in the Heart code, as the workload is proportional to the number of particles instead of the size of the slab. In other phases of the Heart code, such as the FFT, the workload is proportional to the slab size. We would repartition the slab for that phase. Since the grid is in shared memory, repartitioning only involves the change of end points. For higher thread counts, such as in the case for Victoria Falls, 1D partitioning of the grid is not sufficient. There are more threads than the number of y-z planes with particles in them. We used a 2D partitioning in those cases. Particles are first sorted by their X coordinates, followed by the Y coordinate. The grid is first partitioned into slabs along the X dimension, then each slab is partitioned along the Y dimension.

In the spread force phase, grid cells can be updated in one of three ways: direct update on the shared grid, update to a mirror followed by a reduction at the end of the phase, and update on the shared grid with a lock. We want to maximize the number of updates that goes directly on the shared grid without locking. This is the optimization goal for the adaptive implementation. A thread's particle that updates a grid cell owned by the same thread can be done in this

fashion. Updates to the ghost region in the adaptive implementation belong to the second type of updates. This also applies to the mirror replication implementation. The optimization point for the second type of updates is to decrease the cost of the reduction. The reduction cost is directly proportional to the size of the replica. In the mirror replication approach, the replica is always the same size as the size of the slab partition. This may incur reduction cost on grid cells that receive zero updates. This occurs in the Heart code due to the non-uniform particle density. In contrast, the ghost regions in the adaptive implementation are formed dynamically as grid cells receive updates. An extra data structure is used to keep track of the replica boxes. As seen in the memory usage figures, memory footprint size can be significantly reduced using the adaptive approach in comparison to the mirror replication approach.

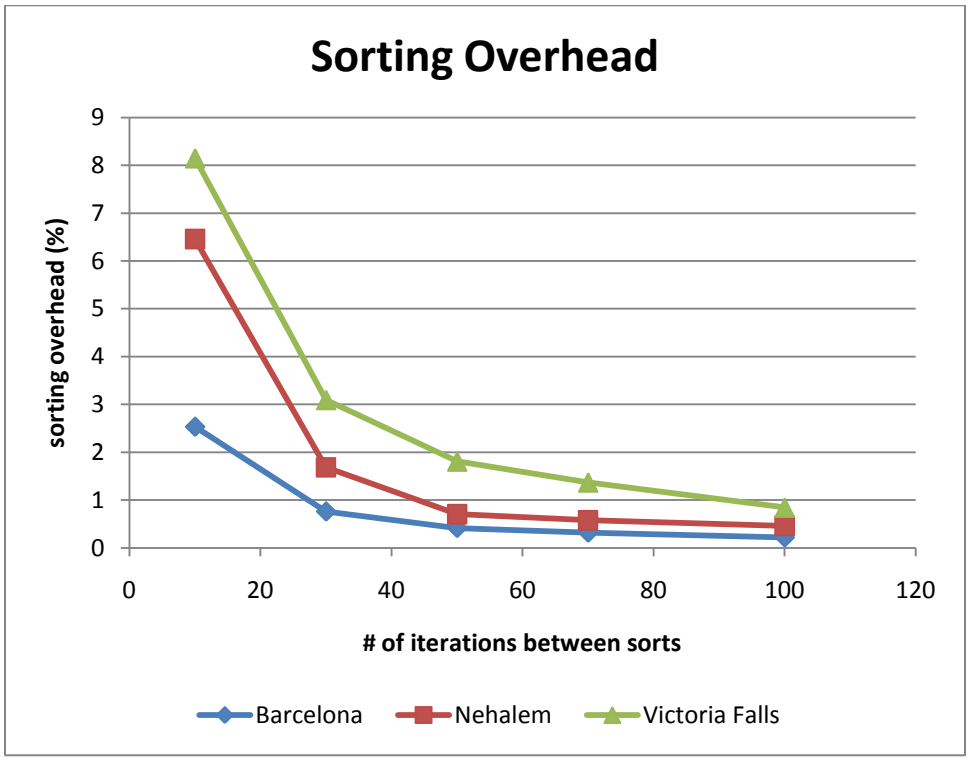


Figure 6.15: Sorting overhead in the adaptive implementation for heart spread force computation. Sorting overhead decreases to below 1% on all three architectures as we increase the number of iterations between sorts to 100.

Figure 6.15 illustrates the overhead of sorting on all three architectures as a function of the number of iterations in between sorts. The experiments use the maximum number of thread contexts on each architecture: 16 threads for Nehalem, 8 threads for Barcelona, and 128 threads for Victoria Falls. As we increase the number iterations in between sorts, the cost of

sorting the particles is amortized. The overhead at 10 iterations is 6.45%, 2.53%, and 8.14% on Nehalem, Barcelona, and Victoria Falls, respectively. At 100 iterations, the overhead decreases to less than 1% on all three architectures. For the Heart code, the particles move slowly, so a few hundred iterations in between sorts would not increase the size of the ghost region significantly. The adaptive implementation relies on sorting to maximize the number of particles that have updates locally. Clearly the performance gains outweigh the overhead of sorting.

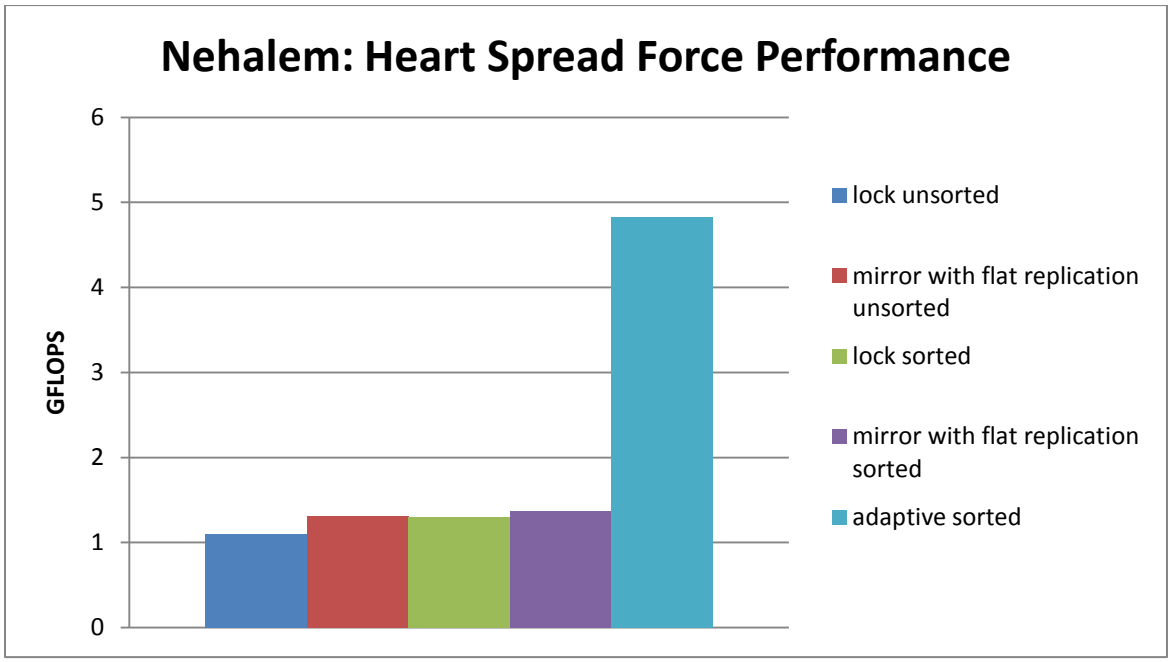


Figure 6.16: Performance summary of different heart spread force implementations on Nehalem using 16 threads.

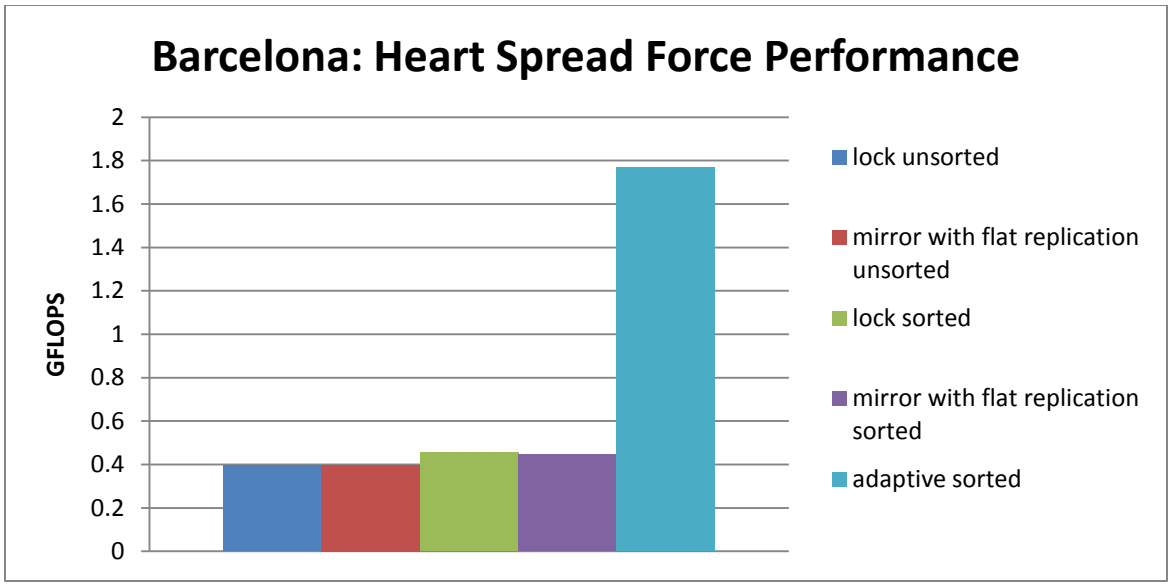


Figure 6.17: Performance summary of different heart spread force implementations on Barcelona using 8 threads.

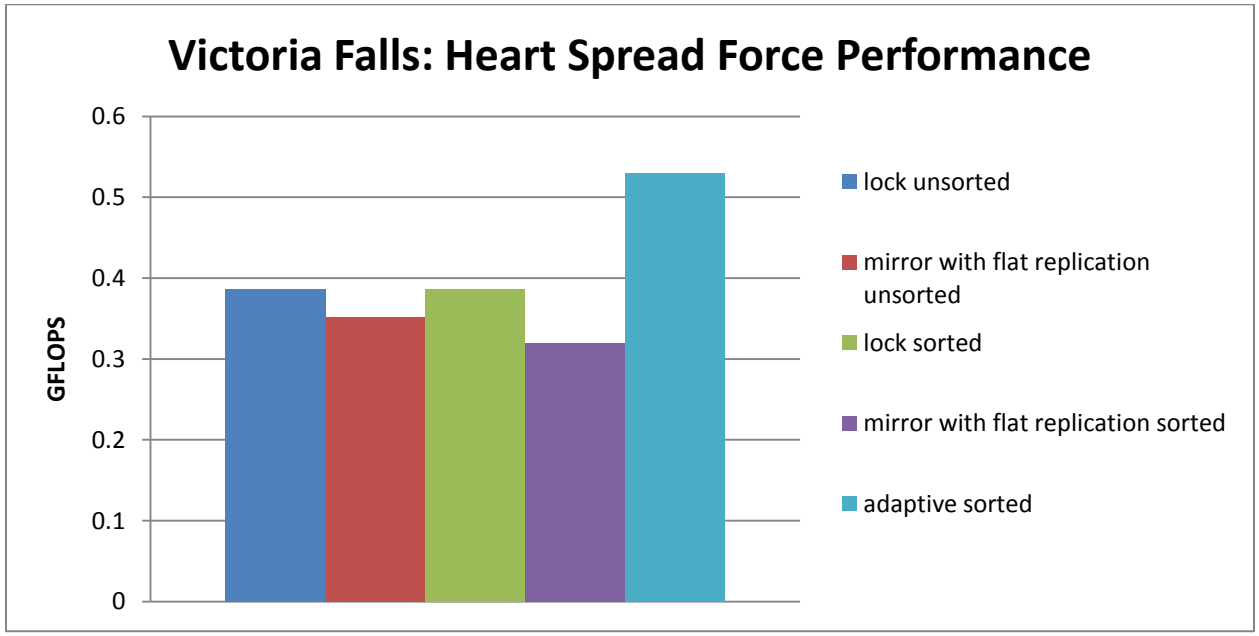


Figure 6.18: Performance summary of different heart spread force implementations on Victoria Falls using 128 threads.

Figure 6.16, Figure 6.17, and Figure 6.18 shows the performance summaries of various heart spread force implementations on Nehalem, Barcelona, and Victoria Falls, respectively. The experiments were done using the maximum number of hardware threads available on each machine. Clearly, the adaptive implementation outperforms all other implementations on all three machines.

We also perform scaling studies for the Heart code on all three architectures. The GFLOPS per thread performance for mirror replication decreases with higher thread counts. Even though each thread gets the same number of particles to work on, the ratio of particles that update to the mirror to the particles that update to the shared grid with locking can be very different among the threads. The threads that get a slab partition that is close to the sides of the grid have a much smaller ratio than the threads with center slab partitions. This causes threads with center slabs to wait at the barrier while threads with side slabs process their particles, since updates to the shared grid with locking is significantly slower than updates to the mirror. Update to mirror is 3, 4, and 1.4 times faster than updates to a shared grid with locking on Nehalem, Barcelona, and Victoria Falls, respectively.

On Nehalem, two-way SMT parallelism is used before threads are mapped to a second core. Similar to the histogram benchmark, two-way SMT yields little benefit in this case as GFLOPS performance per thread drops by a factor of two going from one thread to two threads. For the locking implementation, each update is slower than the other two implementations due to the locking overhead. Lock contention is not a huge issue as the performance of locking is about the same for sorted and unsorted. We find the performance of the inline assembly instructions for atomic update to be twice as fast as using Pthread locks on Nehalem and Barcelona.

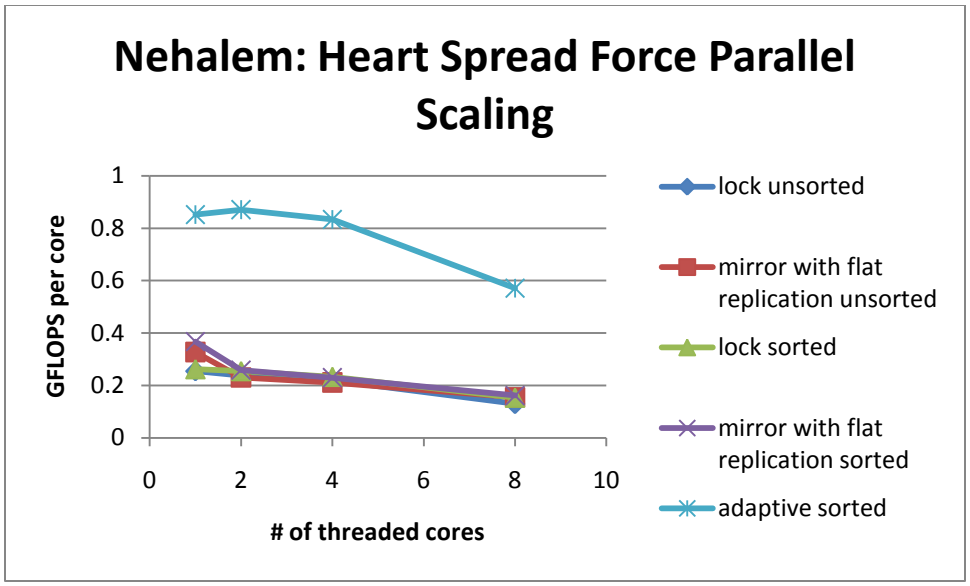


Figure 6.19: Heart spread force per threaded core performance on Nehalem for different implementations

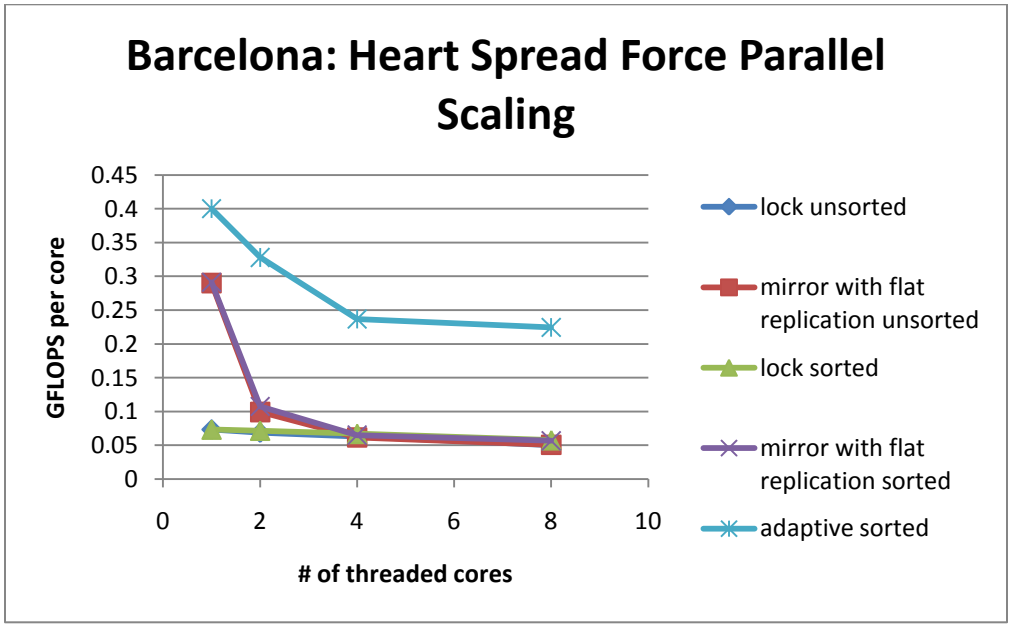


Figure 6.20: Heart spread force per threaded core performance on Barcelona for different implementations

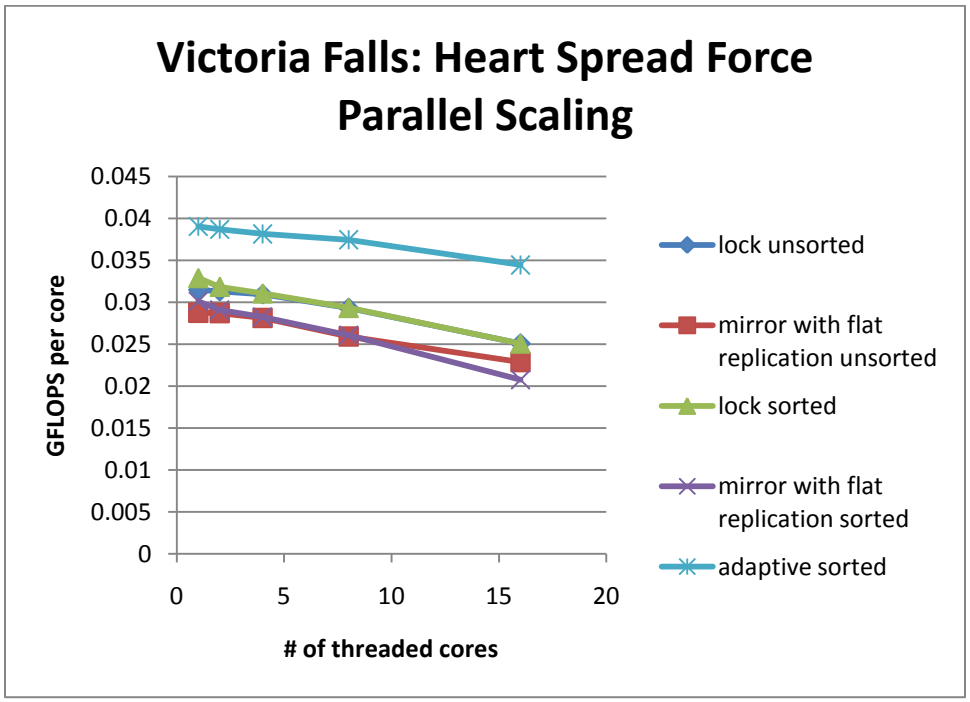


Figure 6.21: Heart spread force per threaded core performance on Victoria Falls for different implementations

The adaptive implementation gives the best performance out of the three implementations. It maximizes the number of updates that can be done directly on the shared grid without locking. The adaptive creation of the ghost region minimized the reduction cost for grid cells that do not receive any updates. The memory footprint size for the adaptive version is within 25% of the locking implementation.

Chapter 7

Performance Debugging

7.1 Introduction

In the previous three chapters, we discuss analysis and optimization techniques for programs with irregular updates on both multicore and cluster architectures. High performance kernels are necessary building blocks for a large parallel application, but they alone do not make the application usable for the end user on large scale machines. There are parts of the program that are less scrutinized for performance that become obstacles for running experiments on large number of processors. These include I/O, initialization, and data distribution. During application development and testing on small number of processors and small input sizes, these problems are typically hidden behind long running kernels. When the application is in production, these problems can no longer be ignored as they are not scalable with the number of processors or input size. This causes longer waiting time for the experiments to finish or not finish at all. In this chapter, we introduce a performance debugging tool for catching these problems earlier in the development cycle in the context of Titanium.

Titanium is a Partitioned Global Address Space language. It combines the programming convenience of shared memory with the locality and performance control of message passing. In Titanium, a thread running on one processor can directly read or write the memory associated with another. This feature significantly increases programmer productivity, since the programmer does not need to write explicit communication calls as in the message passing model. Unfortunately, this is also a significant source of performance bugs. Many unintended small remote reads and writes go undetected during manual code audits, because they look exactly the same as local reads and writes in the program text. Furthermore, these performance bugs often do not manifest themselves until the program is run with large processor configurations and/or large input sizes. This means the bugs are caught much later in the development cycle, making them more expensive to fix.

In this chapter, we describe an automatic communication performance debugging tool for Titanium that can catch this type of bugs using only program runs with small processor configurations and small input sizes. Trends on the number of communication calls are presented to the programmer for each location in the source code that incurred communication during the program runs. Each trend is modeled by a linear function or a power law function in terms of the number of processors or the input problem size. The models can be used to predict communication performance bottlenecks for processor

configurations and problem sizes that have not yet been run. We used the debugging tool on two of the largest Titanium applications and report the bugs that were found using the tool.

7.2 Motivating Example

To illustrate the difficulty of manual performance debugging in a PGAS language like Titanium, we will use a simple sum reduction example in this section. Processor 0 owns a double array. We would like to compute the sum of every element in the array. To spread the workload among the processors, each processor gets a piece of the array and computes the sum for that part. At the end, the partial sums are added together using a reduction.

Two versions of the code are shown in Figure 7.1 and Figure 7.2. The code in Figure 7.1 has a performance bug in it. The two versions are identical except for two lines of code. The loop that computes the actual sum is identical. In the buggy version, each processor only has a pointer to the array on processor 0. `array.restrict(myPart)` returns a pointer to a subsection of `array` that contains elements from `startIndex` to `endIndex`. Each dereference in the foreach loop results in communication to processor 0 to retrieve the value at that array index. Processor 0 becomes the communication bottleneck as all other processors are retrieving values from it.

```
1  double [1d] array;
2  if (Ti.thisProc() == 0){
3     array = new double[0:999];
4  }
5  array = broadcast array from 0;
6  int workload = 1000 / Ti.numProcs();
7  if (Ti.thisProc() < 1000 % Ti.numProcs()){
8    workload++;
9  }
10 int startIndex = Ti.thisProc() * workload;
11 int endIndex = startIndex + workload - 1;
12 RectDomain<1> myPart = [startIndex:endIndex];
13 double [1d] localArray = array.restrict(myPart);
14 double mySum = 0;
15 double sum;
16
17 foreach (p in localArray.domain()) {
18   mySum += localArray[p];
19 }
20 sum = Reduce.add(mySum, 0);
```

Figure 7.1. Sum reduction example with performance bug in it (Version 1)

```
12  RectDomain<1> myPart = [startIndex:endIndex];
13  double [1d] localArray = new double[myPart];
14  localArray.copy(array.restrict(myPart));
15  double mySum = 0;
16  double sum;
17
18  foreach (p in localArray.domain()) {
19    mySum += localArray[p];
20  }
21
22  sum = Reduce.add(mySum, 0);
```

Figure 7.2. Sum reduction example without the performance bug (Version 2)

Figure 7.2 shows the version without the performance bug in it. Each processor first allocates space for the `localArray`, then it retrieves the part of `array` that it needs into `localArray` using one array copy call. The array copy results in one bulk get communication. The subsequent dereferences inside the loop are all local.

Although this is a very simple example, this kind of communication pattern is quite common, especially in the initialization phase of a parallel program, where processor 0 typically processes the input before distributing the workload to the rest of the processors. It is difficult to catch this type of bugs manually in Titanium, since the two versions of the program look very similar. For small processor configurations, the performance degradation may not be noticeable given that the initialization is run only once.

We would like a tool that can alert the programmer to possible performance bugs automatically earlier in the development cycle, when we are only testing the program with small processor configurations and small input sizes. For this example, the number of communication calls at the array dereference in the buggy version can be expressed as `(1-1/p)*size`, where `p` is the number of processors and `size` is the size of the array. If we fix the array size at 1000 elements, then we can see that the number of communication calls at the array dereference varies with the number of processors as in Figure 7.3. The graph shows the actual observed communication calls at the array dereference for 2, 4, and 8 processors along with the predicted curves for both versions of the code.

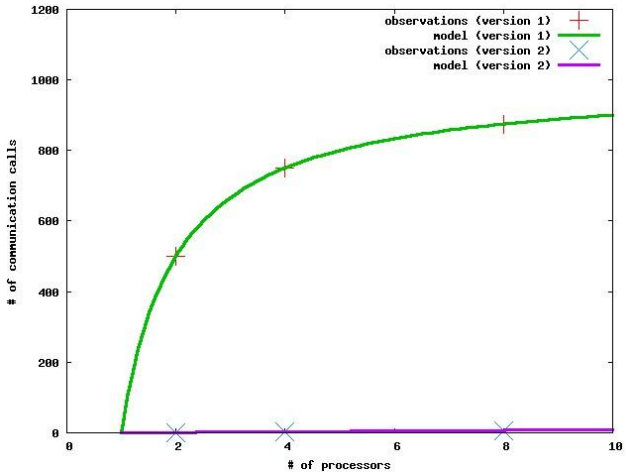


Figure 7.3. The number of communication calls at the array dereference is expressed in terms of the number of processors for a fixed array size of 1000 elements for both versions of the program. The X axis is the number of processors, and the Y axis is the number of communication calls. Version 1 is clearly not scalable. For larger array sizes, the gap between version 1 and version 2 would widen.

In the rest of this chapter, we will describe a tool called `ti-trend-prof` that can present communication trends automatically given only program traces for small processor configurations and/or small input sizes.

7.3 Background

Before getting into the details of `ti-trend-prof`, we will give the necessary background information in this section. This includes a brief introduction on `trend-prof` [30].

7.3.1 `trend-prof`

`trend-prof` is a tool developed by Goldsmith, Aiken, and Wilkerson for measuring empirical computational complexity of programs. It constructs models of empirical computational complexity that predict how many times each basic block in a program runs as a linear or a power law function of user-specified features of the program's workloads. An example feature can be the size of the input. It was previously used on sequential programs for performance debugging.

7.4 Bug Types

In parallel programming, there are many causes for communication performance bugs. This includes excessive amount of communication calls, excessive volume of communication, and load imbalance. Our work so far in `ti-trend-prof` has been focused on finding the first type of bugs automatically. Our framework can be extended to address the other two types of bugs. In Titanium, there are two main causes for excessive amount of communication calls:

1. Remote pointer dereference
2. Distribution of global meta-data

The first case can come up in two situations. One is when a processor has a shallow copy of an object that contains remote references in its fields. Even though the object is in local memory, accessing its field that contains remote reference would result in a round trip of small messages to a remote processor. If the field is accessed frequently during program execution, it can significantly degrade performance. The second situation comes up during workload distribution among processors. In parallel program, it is often the case that one processor does I/O during initialization, and then the workload is distributed among all processors. The motivating example in Section 7.2 fits this description.

The second case comes from distribution of global meta-data. In parallel programs, it is often desirable to have global meta-data available to each processor so that it can find remote objects by following pointers. Each processor owns a list of objects. A naïve way of programming the distribution of meta-data is by broadcasting each pointer individually. This performance bug would not be noticeable when the number of objects is small. Only a large problem size would expose this problem, which is likely to be much later in the development cycle.

In the experimental section, we will show that these types of performance bugs exist in two of the largest Titanium applications written by experienced programmers, and `ti-trend-prof` allowed us to find the bugs automatically within an hour instead of days through manual debugging.

7.5 `ti-trend-prof`

In this work, a new tool called `ti-trend-prof` is developed to combine the use of GASNet trace and `trend-prof` to do communication performance debugging for parallel programs. `ti-trend-prof` takes GASNet trace outputs for small processor configurations and/or small input sizes, and feeds them to a modified version of `trend-prof` that can parse GASNet trace

outputs. The output is a table of trends per Titanium source code location that incurred communication for the input traces.

The number of processors and the input problem size can be used as features. The linear function `a + bx` and the standard power law function with offset `a + bx^c` are used to model the trend at each source code location. The function which minimizes error is picked to be the model. For example, if we fixed the problem size and varied the number of processors, then the trend would tell us how does the number of communication calls change at this location as we vary the number of processors. Similarly, if we fixed the number of processors and varied the problem size, then the trend would tell us how does the number of communication calls change as we vary the problem size. These trends can be used to predict communication performance bottlenecks for processor configurations and input sizes that we have not run yet. This is particularly useful in the beginning of the development cycle, where we do most of the testing on small processor configurations and small inputs. In the table, the trends are first ranked by the exponent, then by the coefficient. Larger values are placed earlier in the table. The goal is to display trends that are least scalable first to the programmer.

In practice, many of the communication patterns can be modeled by the linear function or the power law function. But there are algorithms that do not fall into this category, such as a tree based algorithms or algorithms that change behavior based on the number of processors used. We don't intend to use the linear or power law trends as the exact prediction in communication calls, but rather as an indicator for possible performance bugs. For example, if the number of communication calls at a location is exponential in terms of the number of processors, then `ti-trend-prof` would output a power law function with a large exponent. Although this does not match the actual exponential behavior, it would surely be presented early in the output to alert the programmer.

7.6 Experimental Results

In this section, we show the experimental results on running `ti-trend-prof` on two large Titanium applications: heart simulation and AMR. To obtain the GASNet trace files, the programs were run on a cluster, where each node has a dual core Opteron. We used both cores during the runs. This means that intra-node communication is through shared memory, which does not contribute to communication calls in the GASNet trace counts.

7.6.1 Heart Simulation

The heart simulation code is one of the largest Titanium applications written today. It has over 10000 lines of code developed over 6 years. As the application matures, the focus has been on scaling the code to larger processor configurations and larger problem sizes. The initialization code has remained largely unchanged over the years. Correctness in the initialization code is crucial. But we have not done much performance tuning on the initialization code, since it is run only once in the beginning of execution.

Recently, we had scaled the heart simulation up to 512 processors on a 512^3 problem. On our initial runs, the simulation never got passed the initialization phase after more than 4 hours on the 512 processors. The culprit is in the following lines of code.

```
// missing immutable keyword
class FiberDescriptor{
   public long filepos;
   public double minx, maxx, miny, maxy, minz, maxz;
   ...
}

/* globalFibersArray and the elements in it live on processor 0 */
FiberDescriptor [1d] globalFibersArray;
FiberDescriptor [1d] local localFibersArray;
...
localFibersArray.copy(globalFibersArray);
foreach (p in localFibersArray.domain()){
   FiberDescriptor fd = localFibersArray[p];
   /* Determine if fd belongs to this processor by examining the fields of
fd */
   ...
}
```

Figure 7.4. Fiber distribution code containing a performance bug due to lack of immutable keyword

The programmer meant to add the "`immutable`" keyword to the declaration for the `FiberDescriptor` class. But the keyword was missing. Immutable classes extend the notion of Java primitive type to classes. For this example, if the `FiberDescriptor` were immutable, then the array copy prior to the foreach loop would copy every element in the `globalFibersArray` to the `localFibersArray` including the fields of each element. Without the "immutable" keyword, each processor only contains an array of pointers in `localFibersArray` to `FiberDescriptor` objects that live on processor 0. When each processor other than processor 0 accesses the fields of a `FiberDescriptor` object, a

request and reply message would occur between the processor accessing the field and processor 0. This performance bug is hard to find manually because the source of the bug and the place where the problem is observed are far from each other.

When the processor configuration is small and the number of `FiberDescriptor` objects is small, the effects of this performance bug are hardly observable. Only when we start scaling the application over 100 processors on the 512^3 problem did we notice the problem. The size of the `globalFibersArray` grows proportionately to the problem size of the input. As we increase the number of processors for the same size problem, the number of field accesses to `FiberDescriptor` objects increases linearly. Each processor reads through the entire array to see which fiber belongs to it. Every field access to a `FiberDescriptor` object results in messages to processor 0. At large processor configurations and large problem sizes, the flood of small messages to and from processor 0 becomes the performance bottleneck.

`ti-trend-prof` can catch this bug earlier in the development cycle using only program runs from small processor configurations and small input sizes. It presents the trends in the communication performance both in terms of the number of processors and the input size. Trends are presented for each location in the source code that incurred communication as reflected in the GASNet traces. For a large application such as the heart code, there are many places in the program where communication occurs. In order to present the most interesting results to the user first, trends are sorted first by the exponent followed by the coefficients. Large values get placed earlier in the table. This allows users to see the least scalable locations predicted by the trends first. We also note that the programmer should use `ti-trend-prof` iteratively, each time focusing on the top few locations.

Location	Operation	Feature	Max
FFTfast.ti 8727	Get	41p^2 - 416	198400
FFTfast.ti 8035	Put	41p^2 – 416	198400
MyMailBox.ti 384	Put	9p^2 - 789	404120
MetisDistributor.ti 1537	Get	304690p – 1389867	18330567
FluidSlab.ti 3685	Put	200p	12800
FluidSlab.ti 3725	Put	200p	12800

Table 7.1. Trends output from ti-trend-prof for the heart simulation given the GASNet traces for the 128^3 size problem on 4, 8, 16 and 32 processors

Table 7.1 shows the trends presented by `ti-trend-prof` given GASNet traces for the heart code on 4, 8, 16, and 32 processors for the 128^3 problem. The trend for the performance bug is in red. The trend shows that the number of get calls on line 1537 in the MetisDistributor file is a linear function with a large coefficient. This clearly alarms the programmer since the number of communication calls should be zero at this location if the "immutable" keyword were not missing. Figure 7.5 shows the power law model for the buggy line along with observed data.

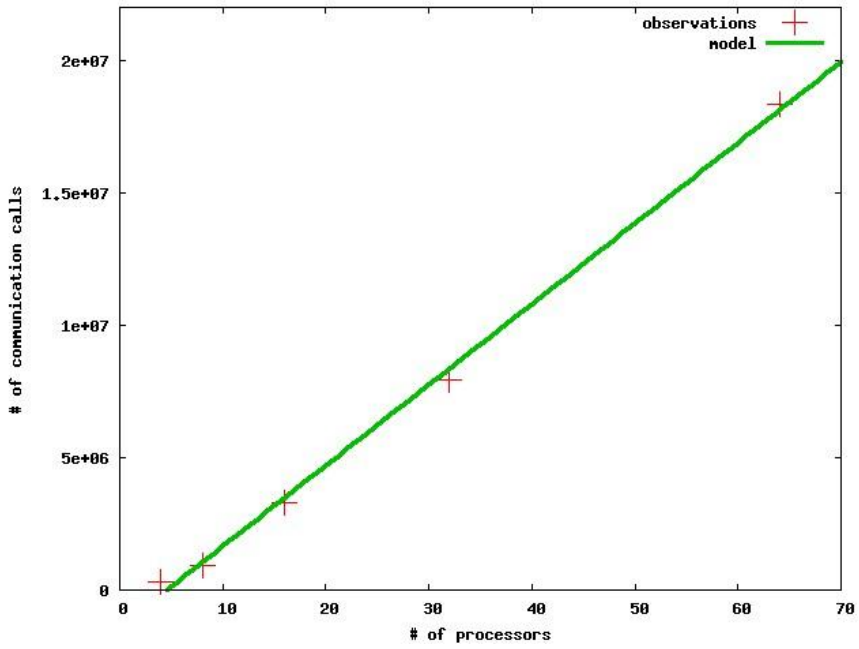


Figure 7.5. Graph of the power law function generated by ti-trend-prof for the buggy line along with actual observations of communication counts. The X axis is the number of processors, and the Y axis is the count of communication calls.

`ti-trend-prof` can find this same bug in another way. Figure 7.6 shows the trend when given GASNet traces for the 32^3, 64^3, and 128^3 size problems on 8 processors. The trend for the performance bug location in terms of the input size also clearly indicates that there is a performance bug here. The number of get calls grows super linearly with the problem size. If the "immutable" keyword were there, there should not be any communication calls for this location. Once the line number is pinned down, it is clear which operation is causing the remote operations. This significantly reduces debugging time, since locating the buggy line of code manually would have taken days. Each run at a large processor count would take hours including program execution time and time waiting in the job queue.

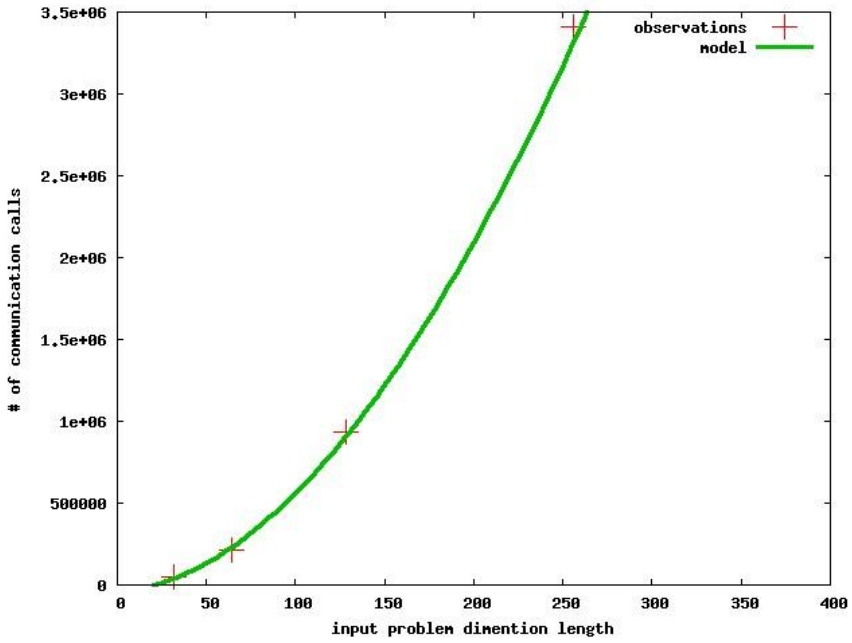


Figure 7.6. Graph of the power law function generated by ti-trend-prof for the buggy line along with actual observations of communication counts. The X axis is the dimension length of the input problem, (dimension length)^3 gives us the input problem size. The Y axis is the communication count.

We also note that not all trends presented by `trend-prof` are performance bugs. For example, the first trend presented in Table 1 represents the communication pattern during the global transpose in the FFT. The global transpose uses an all to all communication pattern, which makes the number of communication calls grow as the square of the number of processors. The trend presented by `trend-prof` confirms this.

7.6.2 Adaptive Mesh Refinement

Adaptive Mesh Refinement (AMR) is another large Titanium application. AMR is used for numerical modeling of various physical problems which exhibit multiscale behavior. At each level of refinement, rectangular grids are divided into boxes distributed among processors. Using `ti-trend-prof`, we were able to find two performance bugs in AMR, where one was known prior from manual debugging and the other was not found previously.

7.6.2.1 Excessive Use of Broadcasts

The first bug appears in the meta-data set up of the boxes at each refinement level. Boxes are distributed among all the processors. But each processor needs to have the meta-data to find neighboring boxes that may live on another processor. Figure 7.7 shows the code for setting up the meta-data. Instead of using array copies to copy the array of pointers from each processor, it uses one broadcast per box to set up the global box array `TA`. For a fixed size problem, the number of broadcasts due to the code in Figure 7.7 is the same regardless of the number of processors. But each processor must wait for the broadcast value to arrive if the broadcast originates from a remote processor. As more processors are added for the fixed size problem, more of the values come from remote processors. Subsequently, each processor performs more waits at the barrier as the number of processors increases, and the total number of wait calls sum over all processors increases linearly as shown in Figure 7.8. If array copies were used, the number of communication calls should only increase by `2p-1` when we add one more processor.

```
/* Meta-data set up*/
for (k=0;k<m_numOfProcs;k++)
    for (j=0;j<(int single)m_layout.numBoxesAt(k);j++)
       TA[k][j]=broadcast TA[k][j] from k;
```

Figure 7.7. Meta-data set up code containing a performance bug due to excessive amount of broadcast calls

Figure 7.8 shows the trend presented by `ti-trend-prof` given the GASNet traces for 2, 4, 6, and 8 processors for the 128^3 problem. It clearly indicates to the programmer that the increase in number of communication calls is larger than expected. Prior to the development of `ti-trend-prof`, it took three programmers to find this bug manually in four days. Similar to the bug in the heart code, the bug was caught late in the development cycle. This performance bug did not become noticeable until we ran the code beyond 100 processors.

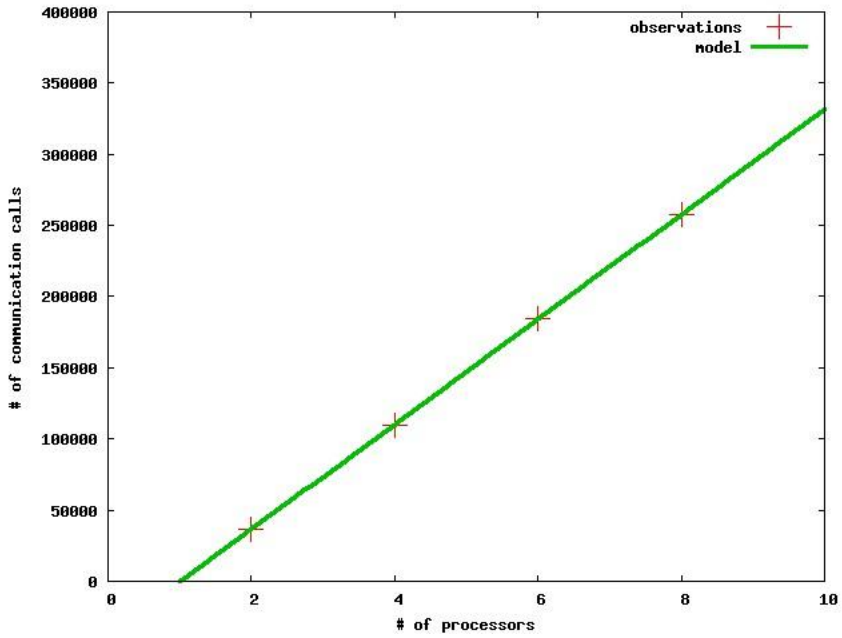


Figure 7.8. Graph of the power law function generated by ti-trend-prof for the excessive broadcast along with actual observations of communication counts. Each processor must wait at the broadcast if the broadcast originates from a remote processor. As the number of processor increases for a fixed size problem, more of the broadcast values come from remote processors.

7.6.2.2 Shallow Copy of Meta-data

After the set up of meta-data, each processor only has a pointer to boxes that live remotely. Whenever it needs to perform operations on the underlying array for the box, it needs to call an accessor method for the box, which incurs communication if the box is remote. The number of calls that require communication increases with the number of processors, because more neighboring boxes become remote as processors are added. `ti-trend-prof` reports that the number of communication calls resulting from the accessor method grows almost as the square of the number of processors. If we had a deeper copy of the meta-pointer, which includes the caching of the pointer to the underlying array, we would avoid a majority of the communication calls at the accessor method. The meta-data for the boxes are reused over many iterations. This bug was not found previous through manual performance debugging.

7.7 Summary

In this chapter, we described a tool called `ti-trend-prof` that can help Titanium programmers to do communication performance debugging automatically. Given only program traces from small processor configurations and/or small input sizes, `ti-trend-prof` provides trends for each source code location that incurred communication. Trends are modeled as a standard power law function with offset. Programmers are alerted to trends with large exponents and coefficients, which correspond to possible communication performance bug in the program. The technique is completely automatic without any manual input from the user.

We used `ti-trend-prof` on two large Titanium applications, and we found three real performance bugs in the code. Two of them were known previously from time consuming manual debugging. The third was unknown prior to the use of the tool. These results show the feasibility of using an automatic tool to find communication performance bugs in PGAS languages, given only the program traces from small processor configurations and small input sizes.

Chapter 8

Related Work

8.1 Irregular Read Access Optimizations

The idea of *inspector executor* optimizations for scientific codes is not new. Walker proposed and implemented the idea of using a pre-computed communication schedule for indirect array accesses to distributed arrays in a Particle-In-Cell application [70]. The idea is widely used in application codes today, where it is programmed manually. Use of the technique with compiler and library support was developed by Berryman and Saltz. The PARTI runtime library [12] and its successor CHAOS [59] provided primitives for application programmers to apply the *inspector executor* optimization on the source level. The same research group provided the dataflow framework to determine where a communication schedule can be generated, where communication operations are placed, and when schedules can be combined [36]. As an experimental result, they manually carried out the optimizations that would have been suggested by the dataflow framework. The ARF [72] and KALI [43] compilers were able to automatically generate *inspector executor* pairs for simply nested loops. Slicing analysis was developed to extend the *inspector executor* paradigm to multiple level of indirection [24]. More recently, the *inspector executor* technique was used to develop runtime reordering of data and computation that enhance memory locality in applications with sparse data structures [60].

Benkner [10] introduces the concept of halos in HPF for the programmers to specify non-local access patterns to distributed arrays, and control the communication associated with these array accesses. Kelp [7] allows the programmers to use the *MotionPlan* object to do *inspector executor* communication. Unlike those two approaches, the optimizations we developed are entirely hidden from the programmer. There have been numerous research works in the area of communication scheduling. Chakrabarti et al. [19] implemented an algorithm for optimizing communication schedules across loops in a global manner in the HPF compiler.

For today's distributed memory machines, the overhead of accessing remote data is usually orders of magnitude higher than local memory accesses. This drastic performance gap has motivated numerous research works that develop compiler algorithms to reduce communication overhead [5, 6, 19, 22, 40, 41, 76]. We have seen fruitful research results in Compilers for data parallel programming languages for communication optimization. For

example, the Stanford SUIF compiler automatically parallelizes and optimizes sequential programs on shared memory multiprocessors [5, 6, 34, 35, 46]. The Polaris system [9, 13] and the PARADIGM compiler [8, 33, 61] are other prominent compiler projects that automatically parallelize Fortran 77 programs. Similarly, a number of optimizations including communication aggregation and pipelining have been implemented for High Performance Fortran [4, 33].

Our work on optimizing irregular read accesses extends the *inspector executor* line of research by looking at the problem of selecting the best communication method. Our work is done in the context of a high level language with a global address space. Our compiler is able to automatically generate code that can accurately choose the best communication method during runtime based on an integrated performance model.

8.2 Irregular Write Access Optimizations

Diniz and Rinard developed the lock elimination algorithm to reduce the synchronization overhead [26]. The algorithm locates computations that repeatedly acquire and release the same lock, then transforms the computations so that they acquire and release the lock only once. The algorithm also increases the sizes of the critical sections, which may decrease the amount of available concurrency. Instead of reducing the number of lock acquires and lock releases, our technique eliminates locking entirely by strength reducing the synchronization to update by owner when possible. This allows most instructions inside of the synchronized region to execute concurrently. Only the updates to the shared memory locations need be executed sequentially.

Rinard developed the optimistic synchronization technique to replace the use of locks for shared memory parallel programs [57]. Instead of using locks for mutual exclusion, it uses the load-linked primitive to retrieve the initial value in an updated memory location. They compute the new value, then use a store conditional primitive to attempt to write the new value back into the memory location. If no other operation wrote the location between the load-linked and the store conditional, the store conditional succeeds. Otherwise, the store conditional fails; the new value is not written into the location; and the operation typically retries the computation. This technique is less effective in a distributed memory setting, since each load-linked and store conditional would require a network roundtrip.

There is much work done in the area of removing unnecessary synchronization in Java programs [3, 14]. In contrast, our work strength reduces synchronizations that are necessary. Barrier elimination and strength reduction are also a well researched topic in the parallel computing community. [64] automatically restructures parallel programs to replace barrier synchronization with less expensive operations, or to remove it entirely. Darte and Schreiber developed a linear time algorithm for optimal barrier placement [23]. Our work looks at a different synchronization construct: synchronized regions.

8.3 Multicore Optimizations

Madduri et al. perform a thorough study of different synchronization and grid partitioning strategies for the GTC (Gyrokinetic Toroidal Code) application on modern multicore processors [48]. The GTC and heart code require a different set of optimizations to achieve optimal performance. The key difference is in the particle distribution in the two codes. For GTC, particles are distributed uniformly throughout the grid in a dense fashion. In contrast, the particles in Heart simulation are sparsely distributed around the center of the grid. Particle density in the heart code is non-uniform, making some effective optimization techniques for GTC less effective in the Heart code. In particular, flat mirror replication in the Heart code is not effective due to non-uniform particle density, making too many useless reductions on grid cells that do not receive any updates. We develop the adaptive replication strategy to address this issue to lower reduction cost. The position of the particles in the Heart code also requires a different grid partitioning strategy than in the GTC code. If we divide the fluid grid in the Heart code into equal sized slabs, many particle to fluid updates would require remote updates (cross socket updates). The memory bandwidth for off socket updates are much less than local updates. We partition the fluid grid in the Heart code according to the sorted order of the particles to address this issue. Particles are sorted in the dimension that the grid is partitioned in.

Our work on PIC multicore optimization serves as a first step toward automated PIC code generation. Automated algorithm tuning and selection have been used in the past in the high performance computing community, especially for numerical libraries. The ATLAS library [71] provides tuned implementations of the BLAS routines [44]. ATLAS uses empirical data gathered at installation time to tune for optimizations such as blocking and loop unrolling. ATLAS performance sometimes exceeds vendor supplied BLAS libraries. Its algorithm tuning is restricted to the linear algebra domain. SPIRAL is another domain specific library that uses automated algorithm tuning and selection [73]. SPIRAL's focus is on signal processing algorithms. Its tuning does not depend on the distribution of the input data.

Rauchwerger, Yu, and Zhang developed adaptive reduction parallelization techniques for shared memory multiprocessor machines [75]. It uses two types of implementations. One approach is replicating iterations that update conflicting memory locations. The second uses variants of the linked-list replicated buffers approach. Linked-list replicated buffer is similar to our replicate and reduce approach, but they did not explore different block sizes for their buffers. It used linear regression to develop predictive models for choosing an implementation. Their work was done at a time where shared memory multiprocessor machines were prevalent. This eliminates the NUMA effects that are addressed in this dissertation.

Brewer used linear regression over three parameters to select the best data partitioning scheme [16]. It used user annotations to determine expected running time. Vuduc used support vector machines to select algorithms for sparse matrix vector multiply [69]. Ganapathi and Datta used machine learning to tune for stencil codes [28]. Pottenger developed the compilation framework for analyzing reduction operations [56]. It did not use predictive models to select between different implementations.

8.4 Performance Debugging

There has been vast amount of work in the area of performance debugging in both sequential programs and parallel programs. For sequential programs, gprof [31] is a widely used tool for estimating how much time is spent in each function. gprof samples the program counter during a single run of the program. Then it uses these samples to propagate back to the call graph during post processing. The key difference is that we use multiple runs of the program to come up with trends that can predict performance problems for processor configurations and/or problem sizes that have not been run. gprof only gives performance information for a single run of the program.

Kluge et al. focus specifically on how the time a MPI program spends communicating scales with the number of processors [42]. They fit these observations to a degree two polynomial, finding a, b, and c to fit y = a+bx+cx^2. Any part of the program with a large value for c is said to parallelize badly. Our work differs in that we can use both the number of processors and the input size as features to predict performance. We have used our tool on large real applications. The experiment in only shows data from a Sweep3D benchmark on a single node SMP. Their technique is likely to have much worst errors when used on a cluster of SMPs. They are modeling MPI time, which would be affected by how many processors are used within a node to run MPI. All processors within a node share resource in communication with other nodes. Furthermore, our target programs are written in a PGAS language instead of MPI, which are much harder to find communication locations manually by looking at the program text.

Vetter and Worley develop a technique called performance assertions that allows users to assert performance expectations explicitly in their source code [68]. As the application executes, each performance assertion in the application collects data implicitly to verify the assertion. In contrast, ti-trend-prof does not require additional work from the user to add annotations. Furthermore, it may not be obvious to the programmer as to which code segment should have performance assertions. ti-trend-prof found performance bugs in code segments where the user didn't think was performance critical. But those performance bugs severely degrade performance only on large processor configurations and large problem sizes, and ti-trend-prof helps the user to identify them by presenting the trends.

Coarfa et al. develop the technique for identifying scalability bottlenecks in SPMD programs by identifying parts of the program that deviates from ideal scaling [20]. In strong scaling, linear speedup is expected. And in weak scaling, constant execution time is expected. Call path profiles are collected for two or more executions on different numbers of processors. Parts of the program that do not meet the scaling expectations are identified for the user. The tool has been developed for Co-Array Fortran, UPC, and MPI. It has an interactive browser for the user to see the contexts for the poor scaling behavior. Our approach differs in the ability to predict poor scaling for large processor configurations based on runs using small number of processors. This allows our tool to be deployed earlier in the development cycle, which lowers the cost for fixing performance bugs.

There are also vast amount of work based on the LogP technique [21]. In particular, Rugina and Schauser simulate the computation and communication of parallel programs to predict their worst-case running time given the LogGP parameters for the targeted machine [58]. Their focus is on how to tune a parallel program by changing communication patterns given a fixed size input.

Chapter 9

Conclusion

In this dissertation, we explore various analyses, optimizations, and tools for irregular data accesses on both multicore and distributed memory cluster architectures. On cluster architectures, we consider both irregular reads and writes, demonstrate how PGAS languages support programming irregular data access problems, and develop optimizations to minimized communication traffic, both in volume and number of distinct events. Our PGAS work is done in Titanium. We explore optimizations for the irregular data access when it appears as a read, using sparse matrix vector multiply as the benchmark, and as a write, using histogram construction and two particle-mesh methods as benchmarks. For sparse matrix vector multiply, the speedup relative to a highly tuned MPI library on a suite of over 20 matrices averages 21% on three different machines, with the maximum speedup of more than 2x. The irregular write optimization yields speedups up to three orders of magnitude over the same code compiled without the optimization on a cluster of Xeons.

On multicore processors, we consider the lower level code generation and tuning problem. We explore performance tradeoffs between various shared update implementations, such as locking, replication of state to avoid collisions, and hybrid versions. The effectiveness of the optimizations is highly dependent on the frequency of updates, the distribution of the accesses across system memory, the likelihood of collisions during updates, and the relative size of the data structures involved. We develop an adaptive implementation that adjusts the shared update strategy based on densities that yields significant speedups and reduced memory footprint. In addition, we develop a performance debugging tool to find scalability problems in large scientific applications earlier in the development cycle.

To summarize the major results of this dissertation:

1. We have developed compiler analysis and optimization for asynchronous parallel updates in the Titanium compiler.

2. We have evaluated the synchronization region optimization on the Heart code spread force, histogram, and particle gravitation benchmarks. The generated code using the optimization achieves speedups of 90x, 120x, and 70x over the sequential code on 128 processors, while the generated code without this optimization does not get speedups above 10x on the same number of processors.

3. We have explored the optimization space for PIC code on modern multicore processors. Our results show that optimizations to maximize the number of local updates are necessary to obtain scalable performance. Simple parallel implementations using locking or full replication yield suboptimal performance.

4. We have developed an adaptive PIC implementation that maximizes the number of local updates. It achieves a speedup up to 4.8x over locking implementation on modern multicore processors.

5. We have developed inspector-executor compiler and runtime optimizations for the Titanium implementation to support programs with indirect read array accesses.

6. We have analyzed the benefits of the automated *inspector-executor* transformation using sparse matrix vector multiplication on a set of matrices from real applications. The speedup relative to MPI on a suite of over 20 matrices averages 21% on three different machines, with the maximum speedup of more than 2x.

7. We have developed a performance debugging tool named ti-trend-prof to find scalability problems in parallel code using multiple program runtime traces. ti-trend-prof is able to predict scalability problems at high processor counts or large input sizes using traces from small number of processors or small input sizes.

8. For two of the largest Titanium applications, Heart code and AMR, ti-trend-prof found multiple scalability bugs within hours instead of days for manual debugging.

Bibliography

[1] A. Aiken and D. Gay, Memory Management with Explicit Regions, Programming Language Design and Implementation, Montreal, Canada, June 1998.

[2] A. Aiken and D. Gay, Barrier inference. In the 25th ACM SIGPLAN-SIGACT Symposiumon Principles of Programming Languages (POPL), pages 342-354, 1998.

[3] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers, Static Analyses for Eliminating Unnecessary Synchronization from Java Programs, Static Analysis: 6th International Symposium, 1999.

[4] R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, 2002.

[5] S. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1993.

[6] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In PLDI 93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, pages 112-125, New York, NY, USA, 1993. ACM Press.

[7] S. Baden, and S. Fink, The Data Mover: A Machine-Independent Abstraction for Managing Customized DataMotion, LCPC, 1999.

[8] P. Banerjee, J. A. Chandy, M. Gupta, E.W. H. IV, J. G. Holm, A. Lain, D. J. Palermo,S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. Computer, 28(10):37-47, 1995.

[9] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. Proceedings of the IEEE, 81(2):211-243, 1993.

[10] S. Benkner, Optimizing Irregular HPF Applications Using Halos, Irregular, 1999.

[11] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. Journal of computational Physics, 53:484-512, 1984.

[12] H. Berryman, and J. Saltz, A manual for PARTI runtime primitives, 1990.

[13] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. IEEE Parallel Distrib. Technol., 2(3):37-47, 1994.

[14] J. Bogda and U. Hazle, Removing Unnecessary Synchronization in Java, ACM SIGPLAN Notices, 1999.

[15] D. Bonachea, GASNet specification. Technical Report CSD-02-1207,University of California, Berkeley, October 2002.

[16] E. Brewer, High-level optimization via automated statistical modeling. SIGPLAN Not., 1995.

[17] D. Callahan, B. Chamberlain, and H. Zima. The Cascade high-productivity language. In 9th International Workshop on High-Level Parallel Programming Modelsand Supportive Environments (HIPS 2004), pages 52-60, April 2004.

[18] S. Chakrabarti, J. Demmel, and K. Yelick, Modeling the benefits
of mixed data and task parallelism, Symposium on Parallel Algorithms and Architectures, 1995.

[19] S. Chakrabarti, M. Gupta, and J. Choi. Global communication analysis and optimization. In SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 68-78, 1996.

[20] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability Analysis of SPMD Codes Using Expectations. PPoPP, 2007

[21] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 112, 1993.

[22] A. Danalis, K. Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In Supercomputing 2005, Nov 2005.

[23] A. Darte and R. Schreiber, A Linear-time Algorithm for Optimal Barrier Placement, tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, 2005.

[24] R. Das, J. Saltz, and R. v. Hanxleden, Slicing analysis and indirect accesses to distributed arrays, Workshop on Languages and Compilers for Parallel Computing, 1993.

[25] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In 18th International Workshop on Languages and Compilersfor Parallel Processing (LCPC), 2005.

[26] P. Diniz and M. Rinard, Synchronization Transformations for Parallel Computing, Concurrency - Practice and Experience, 1999.

[27] D. Fetterly, M. Manasse, M. Najork, Spam, damn spam, and statistics: Using statistical analysis to locate spam web pages, WWW, 2004.

[28] A. Ganapathi and K. Datta, A Case for Machine Learning to Optimize Multicore Performance, HotPar, 2009

[29] D. Gay and A. Aiken, Language Support for Regions, ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, Snowbird, Utah, June 2001.

[30] S. Goldsmith, A. Aiken, and D. Wilkerson, Measuring Empirical Computational Complexity, Foundations of Software Engineering, 2007

[31] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In SIGPLAN Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pages 120-126, New York, NY, USA, 1982. ACM Press.

[32] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In International Conference on Supercomputing, pages 87-96,1993.

[33] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M.Ching, and T. Ngo. An HPF compiler for the IBM SP2. In Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), page 71. ACM Press, 1995.

[34] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In Supercomputing95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing(CDROM), page 49, New York, NY, USA, 1995. ACM Press.

[35] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao,E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. IEEE Computer, 29(12):84-89, 1996.

[36] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J.Saltz, Compiler Analysis for Irregular Problems in FortranD, Workshop on Languages and Compilers for Parallel Computing, 1992.

[37] P. Hilfinger et al. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, November 2001.

[38] A. Kamil and K. Yelick, Concurrency Analysis for Parallel Programs with Textually Aligned Barriers, 18th International Workshop on Languages and Compilers for Parallel Computing, Hawthorne, New York, October 2005.

[39] A. Kamil, Analysis of Partitioned Global Address Space Programs, Master's Report, Computer Science Division, University of California, Berkeley, December 2006.

[40] M. T. Kandemir, A. N. Choudhary, P. Banerjee, J. Ramanujam, and N. Shenoy. Minimizing data and synchronization costs in one-way communication. IEEE Transactionson Parallel and Distributed Systems, 11(12):1232-1251, 2000.

[41] A. Karwande, X. Yuan, and D. Lowenthal. CCMPI: A compiled communication capable MPI prototype for ethernet switched clusters. In ACM SIGPLAN Symposiumon Principles and Practices of Parallel Programming (PPoPP), June 2003.

[42] M. Kluge, A. Knafer, and W. E. Nagel. Knowledge based automatic scalability analysis and extrapolation for MPI programs. In Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lecture Notes in Computer Science. Springer-Verlag.

[43] C. Koelbel, P. Mehrotra, and J. Van Rosendale, Supporting shared data structures on distributed memory machines, Symposium on Principles and Practice of Parallel Programming, 1990.

[44] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Softw., 1979.

[45] B. Liblit and A. Aiken. Type systems for distributed data structures. In the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), January 2000.

[46] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In International Conference on Supercomputing, pages 228-237, 1999.

[47] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mc-Calpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite, 2005. http://www.hpcchallenge.org/pubs/index.html.

[48] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, K. Yelick, Memory-Efficient Optimization of Gyrokinetic Particle-to-Grid Interpolation for Multicore Processors, Supercomputing (SC), 2009.

[49] Matrix Market, http://math.nist.gov/MatrixMarket.

[50] The Message Passing Interface (MPI) standard. http://www.mpi-forum.org/.

[51] R. Mittal and G. Iaccarino, Immersed boundary methods, Ann.Rev. Fluid Mech., vol. 37, pp. 239-261, 2005.

[52] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[53] OpenMP application program interface, 2005.
http://www.openmp.org/drupal/mpdocuments/spec25.pdf

[54] C. Peskin and D. McQueen, A three-dimensional computational method for blood flow in the heart: immersed elastic fibers in a viscous incompressible fluid, Journal of Computational Physics, vol. 81, pp. 372-405, 1989.

[55] G. Pike and P. N. Hilfinger, Better Tiling and Array Contraction for Compiling Scientific Programs, Proceedings of the IEEE/ACM SC2002 Conference.

[56] W. Pottenger, Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs. PhD Thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1997.

[57] M. Rinard, Effective Fine-grain Synchronization for Automatically Parallelized Programs Using Optimistic Synchronization Primitives, ACM Transactions on Computer Systems, 1999.

[58] R. Rugina and K. Schauser. Predicting the running times of parallel programs by simulation. In Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, 1998.

[59] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das,and J. Saltz, Run-time and compile-time support foradaptive irregular problems, Supercomputing, 1994.

[60] M. Strout, L. Carter, and J. Ferrante, Compile-time composition of run-time data and iteration reorderings, Programming Language Design and Implementation, 2003.

[61] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. H. IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In 9th ACM International Conference on Supercomputing, pages 424-433, July 1995.

[62] J. Su and K. Yelick, Array Prefetching for Irregular Array Accesses in Titanium, Sixth Annual Workshop on Java for Parallel and Distributed Computing, Santa Fe, New Mexico, April 2004.

[63] J. Su and K. Yelick, Automatic Support for Irregular Computations in a High-Level Language, 19th International Parallel and Distributed Processing Symposium (IPDPS), 2005.

[64] C. Tseng, Compiler Optimizations for Eliminating Barrier Synchronization, fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, 1995.

[65] R. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid, Official Aztec user's guide: version 2.1, 1999.

[66] UF Sparse Matrix Collection,http://www.cise.ufl.edu/research/sparse/matrices.

[67] The UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.

[68] J. Vetter and P. Worley, Asserting performance expectations, SC, 2002

[69] R. Vuduc, Automatic performance tuning of sparse matrix kernels. PhD thesis, UC Berkeley, 2003.

[70] D. Walker, The Implementation of a Three-Dimensional PIC Code on a Hypercube Concurrent Processor, Conference on Hypercubes, Concurrent Computers, and Application, 1989.

[71] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Computing, 2001.

[72] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani, Distributed memory compiler design for sparse problems,1991.

[73] J. Xiong, J. Johnson, R. Johnson, and D. A. Padua. SPL: A Language and Compiler for DSP Algorithms. PLDI, 2001.

[74] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. Concurrency: Practice and Experience, 10:825-836, 1998.

[75] H. Yu, D. Zhang, L. Rauchwerger, An Adaptive Algorithm Selection Framework, PACT, 2004

[76] Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 199-211, 1998.