Single Program, Multiple Data Programming for Hierarchical Computations

by

Amir Ashraf Kamil

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 Yelick, Chair
Senior Lecturer Paul Hilfinger
Associate Professor Rastislav Bodik
Professor David Wessel

Fall 2012

Single Program, Multiple Data Programming for Hierarchical Computations

Abstract

Single Program, Multiple Data Programming for Hierarchical Computations

by

Amir Ashraf Kamil

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine Yelick, Chair

As performance gains in sequential programming have stagnated due to power constraints, parallel computing has become the primary tool for increasing performance. Parallel computing has long been used in scientific computing, and programmers of the future will likely face many of the same challenges that occur in programming large-scale machines. One such challenge is that of hierarchy: machines are built in a hierarchical fashion, with a wide range of communication costs between different parts of a machine, and applications such as divide-and-conquer algorithms often have hierarchical structure.

Large-scale parallel machines are programmed primarily with the single program, multiple data (SPMD) model of parallelism. This model combines independent threads of execution with global collective communication and synchronization operations. Previous work has demonstrated the advantages of SPMD over other models: its simplicity enables productive programming and avoids many classes of parallel errors, and at the same time it is easy to implement and amenable to compiler analysis and optimization. Its local-view execution model allows programmers to take advantage of data locality, resulting in good performance and scalability on large-scale machines. However, it is a flat model that does not fit well with hierarchical machines or algorithms.

In this dissertation, we introduce the recursive single program, multiple data (RSPMD) execution model. This model extends SPMD with hierarchical, structured *teams*, or groupings of threads. We design RSPMD extensions for the Titanium language, including a hierarchical team data structure and lexically-scoped constructs for operating over teams. We demonstrate that these extensions prevent erroneous use of teams that would result in deadlock. In addition, we present a runtime mechanism for ensuring proper use of both global collective operations and collectives over teams, eliminating more potential sources of deadlock.

As analyzable as SPMD is, we demonstrate that RSPMD can also be analyzed precisely and efficiently. We define a hierarchical pointer analysis for determining which data a pointer can reference, as well as on which threads the referenced data may reside. We then present a series of analyses for computing the set of concurrent statements in both SPMD and RSPMD programs. We show that these analyses improve the results of multiple client analyses, including data-locality and sharing inference, race detection, and memory-model enforcement.

Finally, we present application case studies demonstrating the expressiveness and performance of the RSPMD model. We show that the model enables divide-and-conquer algorithms such as sorting to be elegantly expressed, and that team collective operations increase performance of a conjugate gradient benchmark by up to a factor of two. The model also facilitates optimizations for hierarchical machines, improving scalability of a particle in cell application by 8x, performance of sorting by up to 40%, and execution time of a stencil code by as much as 14%.

Contents

Contents **i**

List of Figures **iii**

List of Tables **vi**

List of Acronyms **vii**

1 Introduction **1**
- 1.1 Thesis Contributions 3

2 Background **5**
- 2.1 The Single Program, Multiple Data Model 5
- 2.2 Titanium 9
- 2.3 The Partitioned Global Address Space Model 10
- 2.4 Collective Alignment 11

3 Language Extensions **13**
- 3.1 Design Goals 13
- 3.2 Team Representation 14
- 3.3 New Language Constructs 17
- 3.4 Implementation 21

4 Alignment of Collectives **23**
- 4.1 The Single Type System 24
- 4.2 Dynamic Alignment 29
- 4.3 Alignment of Team Collectives 33

5 Analysis Background **36**
- 5.1 Machine Model 36
- 5.2 Pointer Analysis 37
- 5.3 Concurrency Analysis 40
- 5.4 Sequential Consistency 41

5.5 Analysis over Teams 44

6 Hierarchical Pointer Analysis **47**
6.1 Language 47
6.2 Concrete Domain 53
6.3 Abstract Domain 53
6.4 Abstract Semantics 54
6.5 Soundness 57
6.6 Algorithm 59
6.7 Locality Inference 60
6.8 Implementation 61

7 Concurrency Analysis **63**
7.1 Flat Concurrency Analysis 63
7.2 Hierarchical Concurrency Analysis 73

8 Evaluation **87**
8.1 Application Case Studies 87
8.2 Dynamic Alignment of Collectives 117
8.3 Pointer Analysis 122
8.4 Combined Analyses 126

9 Related Work **132**
9.1 Hierarchical Computation 132
9.2 Collective Alignment 134
9.3 Pointer Analysis 134
9.4 Concurrency Analysis 135
9.5 Race Detection 135
9.6 Sequential Consistency 135

10 Conclusion **137**

Bibliography **139**

List of Figures

1.1 Hierarchical arrangement of a single Cray XE6 compute node. 2

2.1 Execution of a SPMD program. 6
2.2 Execution of an RSPMD program. 9
2.3 The Titanium thread hierarchy. 10

3.1 Relevant functions from the Titanium `Team` class. 15
3.2 An example of a team hierarchy. 16
3.3 Blocked matrix-vector multiplication. 18

4.1 Difference in execution between strict and weak alignment. 31

5.1 A possible machine hierarchy with four levels. 37
5.2 Example of pointer width with respect to the machine hierarchy team. 38
5.3 A cycle consisting of four accesses in two threads. 42
5.4 Example of a team lattice. 46

6.1 The syntax of the *Ti* language. 49
6.2 Type manipulating functions. 49
6.3 Type checking rules. 50
6.4 Dereferences may require width expansion. 50
6.5 The assignment $y \leftarrow z$ is forbidden. 50
6.6 Sample team hierarchies. 56

7.1 Construction of the interprocedural control-flow graph. 64
7.2 Algorithm 7.1.3 computes the concurrency graph of a program. 66
7.3 The set of code phases for a sample program. 66
7.4 Algorithm 7.1.7 computes the set of all pairs of concurrent expressions in a given program. 68
7.5 Interprocedural control-flow graph for two calls to the same function. 68
7.6 Feasible paths that correspond to unbalanced strings. 69
7.7 Algorithm 7.1.9 uses each method's intraprocedural control-flow graph (G_i) to determine if its exit is reachable from its entry. 71

7.8 Algorithm 7.1.11 computes the set of nodes reachable from the start node through a feasible path. 72
7.9 Algorithm 7.1.13 computes the set of all concurrent expressions that can feasibly occur in a given program. 74
7.10 Example of a concurrency graph. 75
7.11 Concurrency-analysis lattices corresponding to the teams in Figure 6.6. 78
7.12 Possible hierarchical concurrency-analysis lattice corresponding to the teams in Figure 6.6. 79
7.13 Algorithm 7.2.1 computes the concurrency graph of a program by inserting cross edges into its control-flow graph and replacing barrier nodes with barrier edges. 80
7.14 Algorithm 7.2.2 performs a depth-first search on a concurrency graph. 81
7.15 Algorithm 7.2.4 uses each method's intraprocedural concurrency graph to determine under which teams its exit is reachable from its entry. 83
7.16 Algorithm 7.2.5 computes the set of teams under which each node is reachable from the start node through a feasible path. 84
7.17 Algorithm 7.2.7 computes the set of teams under which any pair of expressions can feasibly and concurrently execute in a given program. 86

8.1 A single iteration of matrix-vector multiplication in the original and row team versions of CG. 89
8.2 A single iteration of matrix-vector multiplication in the column team version of CG. . . . 90
8.3 Strong scaling performance of conjugate gradient on a Cray XT4. 92
8.4 Strong scaling performance of conjugate gradient on a Cray XE6. 92
8.5 Strong scaling performance of conjugate gradient on an IBM iDataPlex. 93
8.6 A single iteration of matrix-vector multiplication in the shared-memory version of CG. 94
8.7 Shared-memory sorting algorithm on four threads. 96
8.8 Shared-memory team hierarchy and execution on six threads. 97
8.9 Performance of initial distributed sort on a Cray XT4. 98
8.10 Performance of optimized distributed sort on a Cray XT4. 99
8.11 Performance of optimized distributed sort on a Cray XE6. 100
8.12 Performance of optimized distributed sort on an IBM iDataPlex. 100
8.13 Distribution time on a Cray XE6 when using a single thread per team for communication versus all threads in a team. 101
8.14 Orthographic projection of the heart model. 102
8.15 Sample input sphere, distributed across eight threads. 103
8.16 Sample input torus, distributed across eight threads. 103
8.17 Two-dimensional example of a bounding box containing all of a thread's particles. . . . 105
8.18 Performance of flat particle in cell using shared-memory threads on a NUMA node versus processes that do not share memory. 106
8.19 Performance of various hierarchical particle in cell algorithms on a Cray XE6. 108
8.20 Performance of various hierarchical particle in cell algorithms on an IBM iDataPlex. . 108

8.21 Performance of a 256^3 sphere on a Cray XE6 using the best hierarchical particle in cell algorithm versus the best flat algorithm. 109
8.22 Performance of a 256^3 sphere on an IBM iDataPlex using the best hierarchical particle in cell algorithm versus the best flat algorithm. 110
8.23 Performance of a 512^3 torus on a Cray XE6 using the best hierarchical particle in cell algorithm versus the best flat algorithm. 110
8.24 Performance of a 512^3 sphere on a Cray XE6 using the best hierarchical particle in cell algorithm versus the best flat algorithm. 111
8.25 A five-point stencil in two dimensions. 112
8.26 Weak scaling performance of stencil on a Cray XE6 using OpenMP for shared-memory parallelism. 114
8.27 Weak scaling performance of stencil on an IBM iDataPlex using OpenMP for shared-memory parallelism. 115
8.28 Weak scaling performance of stencil on a Cray XE6 using Pthreads for shared-memory parallelism. 116
8.29 Weak scaling performance of stencil on an IBM iDataPlex using Pthreads for shared-memory parallelism. 117
8.30 Relative collective performance on the SMP machine. 119
8.31 Collective and application performance on the cluster machine in the static case. 119
8.32 Relative collective performance on the cluster machine. 120
8.33 Relative application performance on the cluster machine. 121
8.34 Locality inference results for some Titanium programs. 123
8.35 Sharing inference results for some Titanium programs. 124
8.36 Analysis running time results for some Titanium programs, using various levels of pointer analysis. 125
8.37 Number of data races detected at compile-time. 126
8.38 Performance of barrier and point-to-point variants of CG on a Cray XE6. 128
8.39 Number of memory barriers generated at compile-time. 129
8.40 Average number of memory barriers executed at runtime. 129
8.41 Execution time on an IBM iDataPlex using 32 processors on 32 nodes, compared to a relaxed-consistency version of the code. 130
8.42 Execution time on a Cray XE6 using four processors on a single NUMA node, compared to a relaxed-consistency version of the code. 131

List of Tables

8.1 Fluid size and number of particles for each particle in cell input. 102
8.2 Time for alignment update and check operations on each machine. 122
8.3 Number of operations and calculated overhead for each benchmark. 122

List of Acronyms

AMD	Advanced Micro Devices
AMR	Adaptive mesh refinement
AST	Abstract syntax tree
BSP	Bulk synchronous parallel
CFL	Context-free language
CG	Conjugate gradient
DFS	Depth-first search
FT	Fourier transform
GHz	Gigahertz
HPCS	High-productivity computing systems
HPGAS	Hierarchical partitioned global address space
HPT	Hierarchical place trees
HSPMD	Hierarchical single program, multiple data
HST	Hierarchically well-separated tree
HTA	Hierarchically tiled arrays
IBM	International Business Machines Corporation
ID	Identifier
LQI	Local qualification inference
MG	Multigrid
MIMD	Multiple instruction, multiple data

MPI	Message Passing Interface
NAS	NASA Advanced Supercomputing Division
NERSC	National Energy Research Scientific Computing Center
NUMA	Non-uniform memory access
PGAS	Partitioned global address space
POSIX	Portable operating system interface
QDR	Quad data rate
RSPMD	Recursive single program, multiple data
SC	Sequential consistency
SMP	Symmetric multiprocessor, also shared-memory multiprocessor
SPMD	Single program, multiple data
SQI	Sharing qualification inference
UPC	Unified Parallel C
VLSI	Very-large-scale integration

Acknowledgments

I am grateful to Kathy Yelick for all her help over the last nine years. From the first time I worked with her as an undergraduate teaching assistant to the present, she has always been there to guide and encourage me. I would not have been able to finish without her patience and dedication.

I have spent nearly a decade in the Titanium group and am indebted to everyone who has worked on the project. Dan Bonachea and Jimmy Su have helped me in ways impossible to enumerate, and Paul Hilfinger has always provided valuable insight and feedback. I would also like to thank the members of the Berkeley UPC and GASNet teams, especially Rajesh Nishtala and Yili Zheng, for all their help.

The folks in the VLSI Research Group at Sun Labs were gracious hosts during my time there, particularly Ivan Sutherland and Igor Benko. They provided valuable support and guidance as I struggled to narrow down my dissertation topic. And my many office mates over the years have been a source of both inspiration and distraction as needed, Armando Solar-Lezama, Kaushik Datta, Gilad Arnold, Jimmy, and Rajesh foremost among them.

I would like to thank my friends for keeping me sane, especially Munir, Mahil, and Suhaib, and my family for all their love and support. I'm particularly grateful to Shoaib, whose comradeship on this journey was invaluable.

Finally, all praise is to God, without whom nothing is possible.

The work in this dissertation was supported in part by the Department of Energy under Contracts DE-FC03-01ER25509 and FDDE-FC02-07ER25799, by the National Science Foundation under CNS-0325873 and OCI-0749190, by the California State MICRO Program, and by gifts from Sun Microsystems. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Chapter 1

Introduction

In the past few years, parallel computing has entered mainstream use following the introduction of multicore processors. Previously, computing performance was increased primarily through clock speed scaling, resulting in processors that could execute instructions faster. Unfortunately, this also increased power consumption, leading to problems with heat dissipation at high clock speeds. As raising clock speed could no longer be used to increase performance, vendors resorted to increasing the number of processing cores on each chip, allowing more instructions to complete in a given amount of time through parallel execution. Unlike clock speed scaling, which allows existing software to run faster, increasing parallelism requires programs to be rewritten to take advantage of concurrent execution.

Parallel computing has long been used in scientific computing, where large problem sizes could not be solved on a single processor. As a result, many of the challenges faced in large-scale scientific computing are likely to be encountered in the future by mainstream parallel computing. One specific challenge is that of hierarchical machines. Large scale parallel machines are constructed in a hierarchical fashion, with multiple cores in a processor die, multiple processor dies on a single chip, multiple chips on the same mainboard, multiple boards in a server rack, and multiple server racks in a complete machine. For example, Figure 1.1 diagrams the hierarchical layout of a single compute node in a Cray XE6 machine. This hierarchical organization results in widely varying communication costs between different parts of a machine. Programmers must take into account these costs in order to obtain maximal performance.

Programming large-scale machines is done primarily through the single program, multiple data (SPMD) model. A SPMD program is launched with a fixed number of threads, typically one per core, that execute throughout the program. The SPMD model dominates programming at scale because it encourages "parallel thinking" throughout the program execution, exposing the actual degree of available parallelism. It's local view of execution also naturally leads to good locality and can be implemented by simple, low-overhead runtime systems. The model of communication between threads is orthogonal to choice of control, and both message passing models like the Message Passing Interface (MPI) library [73] and a number of partitioned global address space (PGAS) languages like Unified Parallel C (UPC) [19], Titanium [110], and Co-Array Fortran [77] use the SPMD model by default. Previous work has demonstrated the productivity and performance bene-

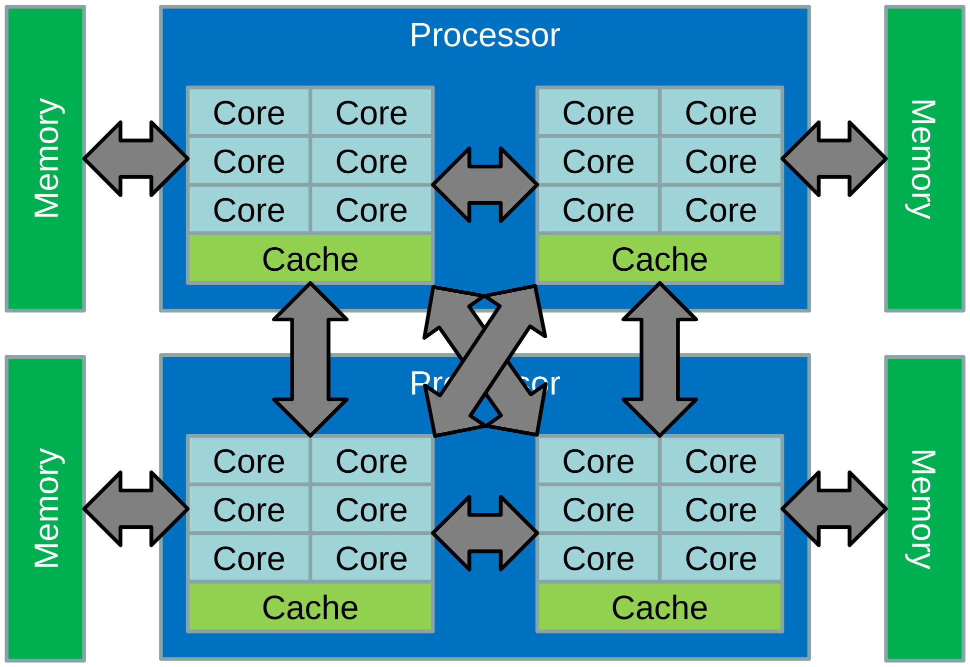


Figure 1.1: Hierarchical arrangement of a single Cray XE6 compute node.

fits of languages that combine SPMD and PGAS in general [109] and Titanium in particular [108].

One of the key features of the SPMD model is the ability to define *collective operations*, which are operations that are performed by all threads together. Collectives allow program synchronization and communication to be specified in a simple and elegant manner, making parallel bugs such as race conditions and deadlock less likely. They also allow global-view operations to be specified in a local-view context. Since all threads must participate in each collective, care must be taken to ensure that such operations are *aligned* on all threads to avoid dependencies between collectives. In the case of the Titanium language, alignment guarantees are provided at compile time but may alternatively be enforced at runtime [53]. Finally, collective operations impose a degree of structure on SPMD programs. The combination of this structure and the fact that all threads execute the same code makes SPMD code amenable to program analyses, enabling many optimizations and correctness tools [60, 65, 66, 52, 51, 54].

While SPMD has proven to be a valuable programming model, its restrictiveness does have drawbacks when it comes to implementing hierarchical algorithms on hierarchical machines. Algorithms that divide tasks among threads or that recursively subdivide do not fit well into a model with a fixed number of threads executing the same code. SPMD programming languages also tend to have a relatively flat machine model, with no distinction between threads that are located nearby on a large-scale machine and threads that are further apart. This lack of awareness of the underlying machine hierarchy results in communication costs that are not transparent to the programmer.

In this dissertation, we address the above shortcomings by introducing the recursive single program, multiple data (RSPMD) model, an extension of the SPMD model with user-defined hierarchical *teams*, subsets of threads that cooperatively execute pieces of code. We demonstrate that this model enables productive programming, provides good performance, eliminates entire classes of parallel bugs, and allows precise compiler analysis.

In order to realize the RSPMD model in the context of the Titanium programming language, we introduce a data structure to represent teams, library functions to facilitate team creation, and language constructs that operate on teams. We demonstrate how to ensure textual alignment of collectives, eliminating many forms of deadlock involving teams. We describe multiple program analyses on RSPMD programs including pointer and concurrency analysis.

We evaluate the RSPMD model through case studies on multiple applications. We demonstrate that hierarchical teams are useful for expressing computations that naturally subdivide, without having to create and destroy logical threads, retaining the advantages of parallel thinking. We also show that hierarchical teams can be used to optimize for the varying communication characteristics of modern, hierarchical parallel machines, achieving significant performance improvements over the flat SPMD model. Finally, we show that dynamic checking ensures alignment of collectives with little performance overhead, and we demonstrate that our program analyses produce useful results for locality and data sharing inference, race detection, and sequential consistency.

1.1 Thesis Contributions

The main contributions of this dissertation are as follows:

- We extend the SPMD model of parallelism to support hierarchical computation, resulting in the RSPMD model. We introduce a powerful library for expressing team hierarchies and language constructs for operating over teams. We show that the design of this team mechanism prevents deadlock that may occur in non-hierarchical schemes for grouping threads, and we implement it in the Titanium language.

- We demonstrate how to enforce textual alignment of collectives using runtime checks, eliminating a major class of bugs in parallel programming. We show how to guarantee alignment in programs written with both the flat SPMD model and the hierarchical RSPMD model. We provide experimental evidence that there is low runtime overhead in checking collective alignment.

- We develop a pointer analysis for programs that use the RSPMD execution model and the hierarchical partitioned global address space (HPGAS) memory model. We show that this analysis improves data locality and sharing inference over constraint-based analyses.

- We develop an efficient analysis for determining the set of potentially concurrent statements in SPMD and RSPMD programs.

- We write a race detector using our pointer and concurrency analyses, proving that they improve precision and significantly reduce false positives.

- We implement a version of the Titanium compiler that provides a sequentially-consistent memory model, using the pointer and concurrency analyses to minimize the runtime cost of doing so. We demonstrate that applications compiled in such a model can achieve performance that is close to that of a relaxed memory model.

- We implement a set of benchmarks using the RSPMD model and its Titanium implementation. We show that the model enables divide-and-conquer algorithms such as sorting to be expressed elegantly. We demonstrate that team collectives provide better performance and productivity than hand-written communication in a conjugate gradient application. We use RSPMD constructs to perform optimizations for hierarchical machines on sorting, particle in cell, and stencil benchmarks. We show that these optimizations improve scalability of particle in cell and performance of sorting and stencil on multiple contemporary platforms.

Chapter 2

Background

In this chapter, we describe the basic language features that we build upon in this dissertation. We start by reviewing the single program, multiple data (SPMD) model of parallelism, one of the most common models used on large-scale machines and the basis for the recursive single program, multiple data (RSPMD) model that we introduce. We then provide a brief overview of the Titanium language that is the vehicle for our work, as well as the partitioned global address space (PGAS) memory model that it uses. Finally, we discuss alignment of collective operations, an important aspect of correctness of large-scale parallel codes.

2.1 The Single Program, Multiple Data Model

The single program, multiple data (SPMD) model of parallelism consists of a set of parallel threads that run the same program. Unlike in dynamic task parallelism, the set of threads is fixed throughout the entire program execution. The threads can be executing at different points of the program, though *collective operations* such as barriers can synchronize the processes at a particular point in the program.

As an example of SPMD code, consider the following written in the Titanium language:

```
public static void main(String[] args) {
  System.out.println("Hello from thread " + Ti.thisProc());
  Ti.barrier();
  if (Ti.thisProc() == 0)
    System.out.println("Done.");
}
```

A fixed number of threads, specified by the user on program start, all enter `main()`. They first print out a message with their thread IDs, or *ranks*, which are unique integers in the range 0 to the number of threads minus one. These messages can appear to the user in any order, since the print statement is not synchronized. Then the threads execute a *barrier*, which prevents them from proceeding until all threads have reached it. Finally, thread 0 prints out another message

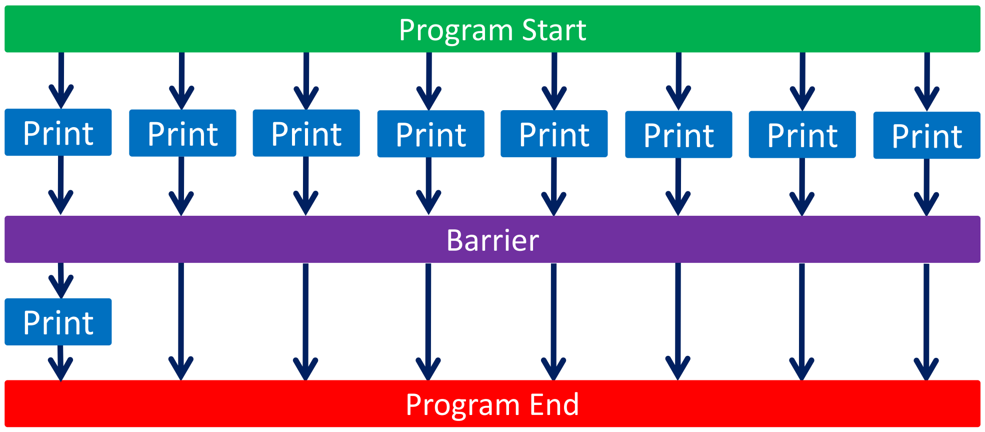


Figure 2.1: Execution of a SPMD program.

that appears to the user after all previous messages due to the barrier synchronization. Figure 2.1 illustrates the execution of this code on eight threads.

Like task-parallel but unlike data-parallel programs, SPMD programs have a *local view* of execution, meaning that each thread is explicitly assigned work. Data parallelism, on the other hand, provides a *global view*, in which there is a single logical thread of control, and the compiler is responsible for distributing work across computational units. In SPMD, collective operations such as a one-to-all *broadcast* or an all-to-all *exchange* provide simple global-view operations in the context of a local-view model and can be used to build larger, more complicated global-view operations.

The set of SPMD languages includes Unified Parallel C (UPC) [19], Co-Array Fortran [77], and Titanium [110]. In addition, though the MPI library [73] supports arbitrary task-parallel code, most MPI programs are written in a SPMD[1] manner, since the added structure of this programming style simplifies the job of the programmer.

SPMD languages occupy a middle ground between task and data parallelism; they are more structured and therefore easier to program and more analyzable than task-parallel languages, but are more flexible than data-parallel languages, allowing expression of irregular applications that is difficult in the data parallel model. Programmer control over work decomposition also generally leads to better performance than the automated decomposition in data parallelism. The following summarizes the advantages of SPMD in more detail:

- **Locality.** Data locality is essential to achieving good performance on large-scale machines, where communication across the network is very expensive compared to computation and local-memory access. Since SPMD has a local view of execution, it allows programmers to write code that concentrates computation on local data and minimizes communication between threads.

[1]SPMD is usually pronounced as "spimdee", so we precede it with the indefinite article "a".

- **Structured parallelism.** The set of threads is fixed throughout computation, exposing the exact degree of parallelism to the programmer. The fact that all threads execute the same program makes it easier to reason about parallel behavior. Global synchronization operations allow programs to be divided into communication and computation phases, reducing the possibility of race conditions, deadlock, and other parallel bugs that are common in the task parallel model. These same features also make it easier for compilers to reason about SPMD code, resulting in more efficient and precise program analyses than in other models.

- **Simple runtime implementation.** Since SPMD has a local view of execution and parallelism is exposed directly to the user, compilers and runtime systems require much less effort to implement than with data parallelism. The latter can require complicated compiler analysis and runtime mechanisms for data distribution and load balancing. In SPMD, these tasks are handled by the programmer.

- **Global operations.** Basic collective operations facilitate global synchronization and communication. More complicated global view operations can also be built using the basic collectives, allowing the tasks of data distribution and load balancing to be offloaded to a library, replicating the productivity benefits of data-parallel languages. For example, a distributed matrix library can provide global interfaces for constructing distributed matrices as well as operating over them. These interfaces would appear to the programmer like any other global operations and would be just as easy to use.

As a result of these advantages, large-scale machines are predominantly programmed using a pure SPMD model, combinations of multiple SMPD components, or combinations of SPMD and shared-memory components.

On the other hand, SPMD in its current form does suffer from some drawbacks. The flat model of parallelism, with only global collective operations, makes it difficult to write hierarchical code. This includes codes with application-level hierarchy, such as divide-and-conquer algorithms, as well as programs optimized for hierarchical machines. For the latter, common practice is to write SPMD code for the distributed portion of a machine and then to use a shared-memory parallel library such as OpenMP [80] for the shared-memory part of a machine. Such a strategy does not take full advantage of a machine, since only a single thread in each shared-memory domain is used for communication. In §8.1, we show that better performance can be obtained in many cases by using multiple threads for communication.

As an example of application-level hierarchy, consider a merge sort written in a task parallel manner.

```
int[] mergeSort(int[] data) {
  int len = data.length;
  if (len < threshold) {
    return sequentialSort(data);
  }
  d1 = fork mergeSort(data[0:len/2-1]);
  d2 = mergeSort(data[len/2:len-1]);
```

```
 8    join d1;
 9    return merge(d1, d2);
10  }
```

The code starts with a single thread and then forks off another thread in each divide step, until the data size reaches a threshold where the overhead of using more threads is greater than the gain from any extra parallelism. The join operation in line 8 waits for the sort to be completed on the forked thread before performing each merge step.

Since SPMD does not allow dynamic creation of new threads, the merge sort algorithm needs to be rewritten to make use of the existing set of threads. This can be done as follows.

```
 1  int[] mergeSort(int[] data, int[] threadIds) {
 2    int len = data.length;
 3    int numThreads = threadIds.length;
 4    if (numThreads == 1) {
 5      return sequentialSort(data);
 6    }
 7    if (threadIds[0:numThreads/2-1].contains(myId)) {
 8      d1 = mergeSort(data[0:len/2-1], threadIds[0:numThreads/2-1]);
 9    } else {
10      d2 = mergeSort(data[len/2:len-1],
11                     threadIds[numThreads/2:numThreads-1]);
12    }
13    barrier(threadIds);
14    if (myId == threadIds[0]) {
15      return merge(d1, d2);
16    }
17  }
```

Here, the code starts with all threads, repeatedly dividing the data and set of threads in half until only one thread remains. Each thread sequentially sorts its data, after which threads recombine to perform the merge step, until all threads reach the top level. The barrier in line 13 performs the same function as the join in the task parallel algorithm, waiting for all relevant threads to finish sorting before the merge operation can occur. Unlike in flat SPMD, however, it should act only on a subset of threads, not all the threads in the program. We refer to such subsets of threads as *teams*. Teams are represented by arrays of thread IDs above; in §3.2, we define a more complex data structure for teams.

Another common example of application-level hierarchy is to divide the initial set of threads among multiple distinct tasks. For example, a climate modeling code may use separate threads to model the ocean, land, and the atmosphere. Each such task may be written independently, requiring them to be composed in a single application. This composition is difficult in flat SPMD, and thread teams can be used to facilitate the process.

Despite the fact that teams are represented by flat vectors in the merge sort algorithm above, there is a hierarchical relationship between teams: starting from the global team, each team is recursively subdivided into two smaller teams. We thus define the recursive single program, multiple

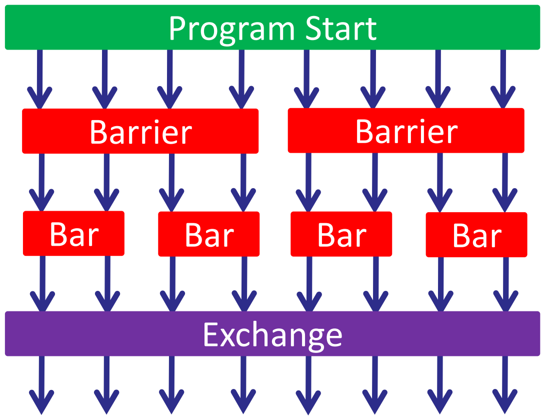


Figure 2.2: Execution of an RSPMD program.

data (RSPMD) model of parallelism as an extension of SPMD that provides hierarchical teams. Figure 2.2 illustrates the execution of an RSPMD program.

In this thesis, we extend the Titanium language to support the RSPMD execution model. Other languages such as UPC are moving towards an execution model based on teams [5], and the GASNet [17] runtime layer used in Titanium now has experimental support for teams and team collectives. Unlike the teams in our work, teams in both UPC and GASNet are non-hierarchical groupings of threads. As we demonstrate throughout this thesis, hierarchical teams provide safety and analyzability advantages over flat teams.

2.2 Titanium

The work in this thesis is done in the context of the Titanium language [110]. Titanium is an explicitly parallel dialect of Java that uses the SPMD model of parallelism and the PGAS memory model described in §2.3. In addition, it provides language features for parallel and scientific programming. These include multidimensional arrays and index spaces, immutable classes, region-based memory allocation [2, 40], C++-style templates, operator overloading, and barrier synchronization. In this report, we discuss only those features that are relevant to our analyses.

The Titanium compiler 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 [17], 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, distributed-memory clusters of uniprocessors or SMPs, and a number of historical supercomputer architectures.

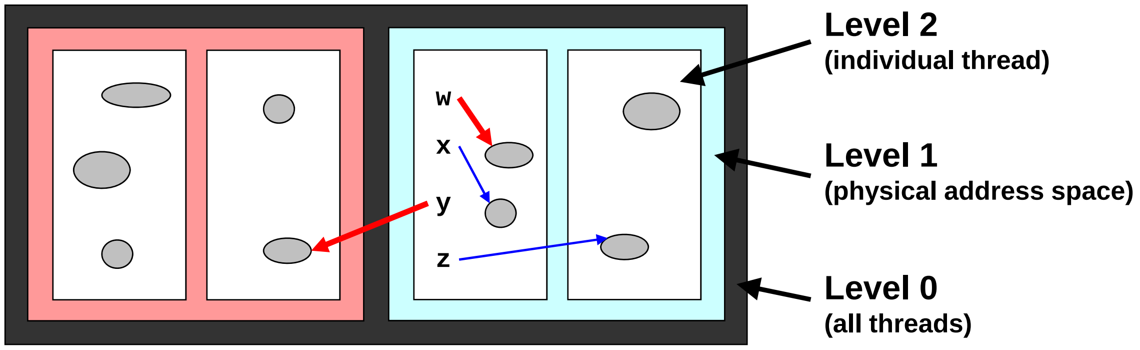


Figure 2.3: The Titanium thread hierarchy. The thin, blue arrows signify node-local and thread-local pointers, while the thick, red arrows designate global pointers. Global pointers may point to local data. Levels are numbered by increasing degree of locality.

2.3 The Partitioned Global Address Space Model

The SPMD execution model specifies the mechanism of *control* in a parallel program; the mechanism of *communication* is orthogonal to that of control. One important communication mechanism is the *partitioned global address space* (PGAS) model, which allows any thread to directly access memory on other threads. At runtime, two threads may share the same physical address space, in which case such an access is done directly using load and store instructions, or they may be in distinct address spaces, in which case the global access must be translated into communication using a library such as GASNet.

As an example, consider the following Titanium code:

```
int[] local mydata = { ... };
int[] data0 = broadcast mydata from 0;
for (int i = 0; i < data0.length; i++)
  ... data0[i] ...
```

In this code, each thread creates an integer array in its own memory space. Thread 0 then broadcasts a pointer to its array to the other threads, which can then access elements of thread 0's array, albeit with a possible performance penalty.

As can be seen in the example above, PGAS languages expose some degree of memory hierarchy to the programmer by virtue of the partitioned address space. In Titanium, pointers can be *thread local*, corresponding to level 2 in Figure 2.3, *node local*, corresponding to level 1, or *global*, corresponding to level 0. Thread-local pointers can only address data on the same thread[2], node-local pointers can only reference data in the same physical address space, and global pointers can point to any object in the program. By default, pointers in Titanium are global, and the **`local`** qualifier specifies that a pointer is node local. There is no specific qualifier for thread-local point-

[2]Thread-local pointers in Titanium are actually only used by program analysis and are not exposed in the type system.

ers, though pointers qualified by the **`nonshared`** keyword are guaranteed to be thread local. Other PGAS languages such as UPC only have two levels of hierarchy.

The hierarchical partitioned global address space model (HPGAS) is an extension of PGAS to an arbitrary machine structure with an arbitrary number of levels. It can be used to represent the memory hierarchy on a single thread, as well as hierarchical network communication costs. Previously, we used the HPGAS model to construct a pointer analysis for Titanium [54]. In §6, we extend the pointer analysis to work on hierarchical teams.

Instead of a partitioned global address space, a unified global address space can be provided to the programmer. Such a memory model, however, does not expose any locality to the user. A global pointer in the PGAS model provides a hint to the programmer that communication may be required; a unified model would not provide such a hint, making performance more opaque to the user. This in turn could make it more difficult to write code that performs well.

Another alternative to PGAS is message passing, in which communication is done by exchanging messages between threads, as is used in the Message Passing Interface (MPI) [73]. Both sender and receiver explicitly participate in communication, requiring significant programmer effort to align the two participants and avoid deadlock. The PGAS model, on the other hand, requires the explicit participation of only the initiator of communication, and thus is simpler to program than message passing.

Previous work has shown that PGAS languages in general [109] and Titanium specifically [108] provide an excellent combination of application performance and programmer productivity, justifying our decision to pursue hierarchical extensions to parallel programming in the context of the Titanium language.

2.4 Collective Alignment

One of the key features of the Titanium language is that it guarantees proper usage of collective operations through textual alignment. Collectives are *textually aligned* if all threads execute the same textual sequence of collective operations, and they agree on all control-flow decisions that affect execution of collectives. For example, the code below violates textual alignment, since different threads take different branches; not only do they not agree on control flow affecting collectives, they reach different textual instances of a collective.

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

Discussions with parallel application experts indicate that most applications do not contain unaligned collectives, and most of those that do can be modified to do without them. Our own survey of eight NAS Parallel Benchmarks [9] using MPI demonstrated that all of their collectives are textually aligned.

Misalignment of collectives can result in deadlock, such as in the following code:

```
1 if (Ti.thisProc() % 2 == 0) // even threads
2   Ti.barrier();
3 else // odd threads
4   y = broadcast x from 0;
```

In this code, threads with even ID wait at the barrier in line 2, while threads with odd ID wait at the broadcast in line 4. Neither operation can complete until all threads have reached it, resulting in deadlock. While this example may be contrived, more complicated cases can occur in practice with much more subtle errors.

Titanium's type system statically ensures that global collective operations are textually aligned by making use of *single-valued* expressions that are semantically the same on all threads [1]. We describe this type system in detail in §4. Unfortunately, the type system requires significant programmer burden and is difficult to extend to collectives over teams. Instead, we introduce in §4 a dynamic scheme that guarantees textual alignment at runtime.

Chapter 3

Language Extensions

The recursive single program, multiple data (RSPMD) model of parallelism consists of a fixed set of cooperating threads that can be recursively subdivided into smaller groups of threads. In this chapter, we define language extensions for Titanium to implement the RSPMD model. We start be discussing the primary design goals for the language extensions. We then present new data structures and language constructs for RSPMD. Lastly, we provide an overview of how the extensions are implemented in the Titanium compiler and runtime.

3.1 Design Goals

In designing the new additions to the Titanium language, we had a few goals in mind for the extensions to satisfy: safety, flexibility, composability, support for collectives, performance, and analyzability.

1. **Safety.** Team implementations in other SMPD languages and frameworks do not generally impose any restrictions on their use. This can lead to circular dependencies in team operations, resulting in deadlock. For example, a set of threads may attempt to perform a collective operation on one team, while other threads attempt to perform a collective operation on a different team; if the two teams overlap, then this situation results in deadlock. The following code illustrates this problem:

```
Team t1 = new Team(0:7);
Team t2 = new Team(0:3);
if (myId == 0) {
  barrier(t1);
} else {
  barrier(t2);
}
```

In this code, **barrier**(`t`) executes a barrier operation over team `t`. Since threads 0 through 3 are in both `t1` and `t2`, thread 0 waits for threads 1 through 3 to arrive at the barrier in line 4, while they in turn wait for thread 1 to arrive at the barrier in line 6. Deadlock is the result.

While the problem is easy to spot in the code above, more complicated examples can be constructed that contain less obvious circular dependencies. The Titanium team extensions should prevent such dependencies, as well as ensure that team collectives are textually aligned on all threads in the relevant team, as is done for existing global collectives.

2. **Flexibility.** A trivial solution to the safety problem would be to restrict each thread to be a member of a single, fixed team throughout program execution, preventing circular dependencies between teams. However, many applications make use of different thread groupings at different points in the program, such as a matrix-vector multiplication that requires both row and column teams. The team mechanism should be flexible enough to support such cases while still providing safety guarantees.

3. **Composability.** Existing code running in the context of a particular team should behave as if the entire world consisted of just the threads in that team, with thread ranks as specified by the team. This is to facilitate composition of different tasks, so that a subset of threads can be assigned to each of them. At the same time, the team mechanism should make it possible to interact with threads outside of an assigned team if necessary.

4. **Support for collectives.** One of the key features of the SPMD programming model is the ability of threads to communicate and synchronize through collective operations, such as reductions and barriers. Without support for collective operations over teams, users would have to hand-write their own implementations, requiring extensive development time and resulting in suboptimal performance. We describe such an example in §8.1.2.

5. **Performance.** Team operations should not adversely affect application performance. This requires that team usage operations, which may be invoked many times throughout an application run, be as lightweight as possible, even at the expense of team creation operations that are called much less frequently.

6. **Analyzability.** The structure of SPMD code makes it more amenable to analysis than other models of parallelism, enabling optimizations and correctness tools that take advantage of analysis results. RSPMD code should similarly be easy to analyze, and the team extensions should avoid making it harder to reason about RSPMD programs.

3.2 Team Representation

In order to represent a team hierarchy, we introduce a new `Team` object, as shown in Figure 3.1. A `Team` represents a group of threads and contains references to parent and child teams, resulting in a hierarchy of teams. Like MPI or GASNet groups, `Team` objects specify team structures separately from their usage; this is useful when a program uses multiple different team structures or repeatedly

```
public class Team {
  /** Create team with all threads in currently executing team. */
  public Team();
  /** Returns the ith child of this team. */
  public Team child(int i);
  /** Number of child teams. */
  public int numChildren();
  /** Rank of this team in its parent. */
  public int teamRank();
  /** Number of threads in this team. */
  public int size();
  /** The child team containing the calling thread. */
  public Team myChildTeam();
  /** Split team into n equally-sized subteams, with threads retaining
   * relative ranks. */
  public void splitTeam(int n);
  /** Split team into n child teams, with threads assigned to subteams
   * in block cyclic order. */
  public void splitTeamBlockCyclic(int n, int sz);
  /** Split team into the given subteams, with ranks specified
   * relative to this team. */
  public void splitTeamRelative(int[][] teams);
  /** Collective split operation. Assigns threads to subteams
   * according to color and the given rank relative to other threads.
   */
  public single void splitTeamAll(int color, int relrank);
  /** Collective split operation. Divides threads into subteams of
   * threads that share memory, with the given relative rank. */
  public single void splitTeamSharedMem(int rel);
  /** Collective operation. Constructs a new team in which each
   * subteam consists of a single thread from each subteam of this
   * team. */
  public single Team single makeTransposeTeam();
  /** Initialize runtime structures required by this team and run
   * consistency checks. */
  public single void initialize(boolean check);
}
```

Figure 3.1: Relevant functions from the Titanium `Team` class.

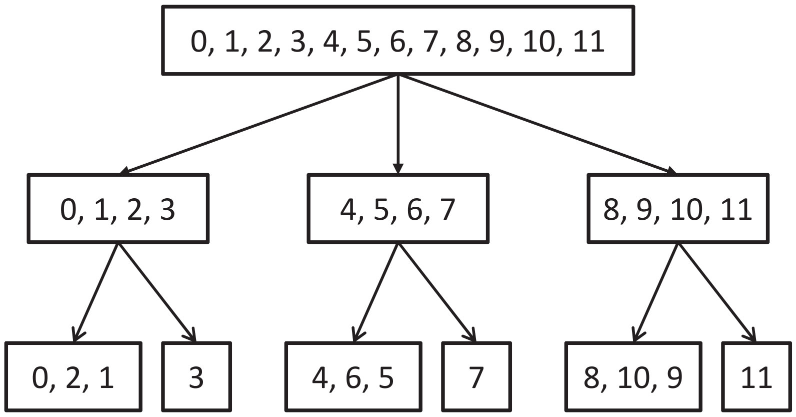


Figure 3.2: An example of a team hierarchy.

uses the same structure, as in §8.1.2, and also allows team data structures to be manipulated as first-class objects.

Knowledge of the physical layout of threads in a program allows a programmer to minimize communication costs, so a new method `Ti.`**`defaultTeam`**() returns a special team that corresponds to the mapping of threads to the machine hierarchy. Currently, it merely divides threads into groups that share memory, though future use of the *hwloc* library [84] can provide a more representative layout. The invocation `Ti.currentTeam`() returns the current team in which the calling thread is participating.

Figure 3.2 shows the team hierarchy created by the following code, when there are a total of twelve threads:

```
Team t = new Team();
t.splitTeam(3);
int[][] ids = new int[][] {{0, 2, 1}, {3}};
for (int i = 0; i < t.numChildren(); i++)
  t.child(i).splitTeamRelative(ids);
```

Each box in the diagram corresponds to a node in the team tree, and the entries in each box refer to member threads by their global ranks.

The code above first creates a team consisting of all the threads and then calls the **`splitTeam`** function to divide it into three equally-sized subteams of four threads each. It then divides each of those subteams into two uneven, smaller teams. The **`splitTeamRelative`** call divides a team into subteams using IDs relative to the parent team. In this case, each child u of team `t` is split into two smaller teams, with threads 0, 2, and 1 of u assigned to the first subteam and thread 3 of u assigned to the second. This behavior allows the same code to be used to divide each of the three children of `t`, which would not be the case if **`splitTeamRelative`** used global IDs.

The `Team` class provides a few other ways of generating subteams, as shown in Figure 3.1. In addition, it includes numerous functions to query team properties, a sample of which are also shown in Figure 3.1. For example, the `teamRank()` function returns the rank of a team in its parent, which can be used to write code that is conditional on a team's rank.

Once a team has been created, the programmer must call the `initialize` method before using the team in the constructs introduced below. It is a collective operation that performs the runtime setup needed by a team and checks team consistency across threads. In our current implementation, this initialization is separate from team creation, allowing a user to construct a team on a single thread and then broadcast it to the others. Those threads then must create local copies since `Team` objects contain thread-specific state. So far, we have not found this to be a useful feature, and we may remove this functionality in order to combine team creation and initialization.

3.3 New Language Constructs

In designing new language constructs that make use of teams, we identified two common usage patterns for grouping threads: sets of threads that perform different tasks and sets of threads that perform the same operation on different pieces of data. We introduce a new construct for each of these two patterns.

3.3.1 Task Decomposition

In task parallel programming, it is common for different components of an algorithm to be assigned to different threads. For example, a climate simulation may assign a subset of all the threads to model the atmosphere, another subset to model the oceans, and so on. Each of these components can in turn be decomposed into separate parts, such as one piece that performs a Fourier transform and another that executes a stencil. Such a decomposition does not directly depend on the structure of the underlying machine, though threads can be assigned based on machine hierarchy.

Task decomposition can be expressed through the following *partition* statement that divides the current team of threads into subteams:

`partition``(`T`) {` B_0 B_1 `...` B_{n-1} `}`

A `Team` object (corresponding to the current team at the top level) is required as an argument. The first child team executes block B_0, the second block B_1, and so on. It is an error if there are fewer child teams than partition branches, or if the given team arguments on each thread in the current team do not have the same description of child teams. If the provided team has more than n subteams, the remaining subteams do not participate in the partition construct. Once the partition is complete, threads rejoin the previous team.

As a concrete example, consider a climate application that uses the team structure in Figure 3.2 to separately model the ocean, the land, and the atmosphere. The following code would be used to divide the program:

```
partition(t) {
  { model_ocean(); }
  { model_land(); }
```

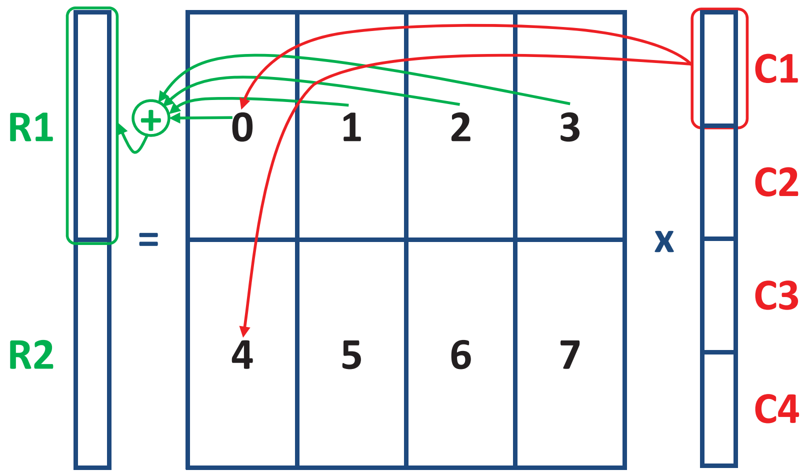


Figure 3.3: Blocked matrix-vector multiplication.

```
    { model_atmosphere(); }
  }
```

Threads 0 to 3 would then execute `model_ocean`(), threads 4 to 7 would run `model_land`(), and threads 8 through 11 would model the atmosphere.

Since partition is a syntactic construct, task structure can be inferred directly from program structure. This simplifies program analysis and improves understandability of the code.

3.3.2 Data Decomposition

In addition to a hierarchy of distinct tasks, a programmer may wish to divide threads into teams according to algorithmic or locality considerations, but where each team executes the same code on different sets of data. Such a data decomposition can be either machine dependent or required by an algorithm, and both the height and width of the hierarchy may differ according to the machine or algorithm.

Consider the matrix-vector multiplication depicted in Figure 3.3, where the matrix is divided in both dimensions. In order to compute the output vector, threads 0 to 3 must cooperate in a reduction to compute the first half of the vector, while threads 4 to 7 must cooperate to compute the second half. Both sets of threads perform the same operation but on different pieces of data.

A new *teamsplit* statement with the following syntax allows such a data-driven decomposition to be created:

`teamsplit`(T) B

The parameter T must be a `Team` object (corresponding to the current team at the top level), and as with partition, all threads must agree on the set of subteams. The construct causes each thread to execute block B with its current team set to the thread's subteam specified in T, so that thread ranks and collective operations in B are with respect to that subteam. As mentioned above, each subteam also has a rank, which can be used to determine the set of data that the subteam is to operate on.

As an example, consider a reduction over the rows of a matrix, as in the following code:

```
  teamsplit(t) {
    Reduce.add(data[t.myChildTeam().rank()], myData);
  }
```

The reduction executes over the current team inside the teamsplit on each thread, which is its associated child team of `t`. As a result, data from threads 0 to 3 are reduced to produce a result for team 0, and data from threads 4 to 7 are combined into a result for team 1.

3.3.3 Common Features

Both the partition and teamsplit constructs are lexically scoped, changing the team in which a thread is executing within that scope. This implies that at any point in time, a thread is executing in the context of exactly one team (which may be a subteam of another team and have child teams of its own). Given a particular team hierarchy, entering a teamsplit or partition statement moves one level down in the hierarchy, and exiting a statement moves one level up. Statements can be nested to make use of multi-level hierarchies, and recursion can be used to operate on hierarchies that do not have a pre-determined depth. Consider the following code, for example:

```
public void descendAndWork(Team t) {
  if (t.numChildren() != 0)
    teamsplit(t) {
      descendAndWork(t.myChildTeam());
    }
  else
    work();
}
```

This code descends to the bottom of an arbitrary team hierarchy before performing work. A concrete example that uses this paradigm is the merge sort in §8.1.3.2.

In order to meet the composability goal of §3.1, the thread IDs returned by `Ti.thisProc()` are now relative to the team in which a thread is executing, and the number of threads returned by `Ti.numProcs()` is equal to the size of the current team. Thus, a thread ID is always between 0 and `Ti.currentTeam().size()`-1, inclusive. A new function `Ti.globalNumProcs()` returns the number of threads in the entire program, and `Ti.globalThisProc()` returns a thread's global rank.

Collective communication and synchronization now operate over the current team. Both the partition and the teamsplit construct are also considered collective operations, so they must be textually aligned in the program. The combination of the requirement that all threads must agree on the set of subteams when entering a partition or teamsplit construct, lexical scoping of the constructs, and textual collective alignment ensure that no circular dependencies exist between different collective operations. In §4.3, we describe how the first and last properties are enforced.

3.3.4 Discussion

It may be apparent that the partition statement can be implemented in terms of teamsplit, such as the following:

```
teamsplit(t) {
  switch(t.myChildTeam().teamRank()) {
  case 0:
    model_ocean();
    break;
  case 1:
    model_land();
    break;
  case 2:
    model_atmosphere();
  }
}
```

While this is true, we decided that an explicit construct for task decomposition is cleaner and more readable than the combination of teamsplit and branching. The two constructs also differ with respect to the superset operations described below.

3.3.5 Superset Operations

By design, the partition and teamsplit constructs require a user to exit or enter a construct to move up or down a team hierarchy or to use multiple team hierarchies. We suspect that it may be useful, however, to be able to temporarily move up one or more levels in a team hierarchy without exiting a partition or teamsplit, though we have yet to find concrete examples where this is the case. Nevertheless, our implementation contains a *superset* statement that ascends the team hierarchy within a specified lexical scope:

superset (i) B

This results in execution of block B in the context of the team that is i levels up from the enclosing team, i.e. that team's ith ancestor. As an example, consider the following code:

```
teamsplit(t) {
  Reduce.add(data[t.myChildTeam().rank()], myData);
  superset(1) {
    Ti.barrier();
  }
  ...
}
```

Inside the teamsplit, threads execute as members of their respective subteams of `t`, so the reduction is over these teams. If threads read data from another team later in the teamsplit, then a global synchronization is necessary to ensure that the reductions have completed on all threads. The superset operation accomplishes this by walking up one level to the global team and performing a barrier.

A superset operation is considered to be a collective operation in the enclosing team as well as its i ancestors. Since teams in a partition execute different code, the i enclosing team statements must all be teamsplits in order to conform to textual collective alignment. It is an error if all threads do not execute the superset in any of the i ancestral teams, if they differ on the value of i, or if the enclosing team has fewer than i ancestors. Superset statements may be nested, but they may not contain any teamsplit or partition statements.

We anticipate that the most likely use of a superset operation will be to perform a collective, such as a barrier, at a higher level in the team hierarchy. As such, we have implemented versions of many collective operations that operate at higher levels without requiring an explicit superset construct. For example, the call `Ti.`**`barrier`**(1) executes a barrier on the parent team. Of course, such operations must meet the same requirements as the superset construct itself.

3.4 Implementation

We have implemented hierarchical team constructs on top of GASNet teams, which are flat groupings of threads. Each node in a team hierarchy is associated with a separate GASNet team, which is created when a team is initialized. We avoid creating unnecessary duplicate GASNet teams by caching them at the Titanium level and checking whether or not there is an existing GASNet team corresponding to a particular Titanium subteam.

The lexical teamsplit, partition, and superset operations are implemented in the compiler as calls to the following library functions:

```
single static TeamHandle checkedSplit(Team t, boolean isTeamsplit);
single static TeamHandle split(Team t, boolean isTeamsplit);
single static void unsplit(TeamHandle th);
```

Upon entering a new team context, `checkedSplit` or `split` is called, depending on whether or not the user has enabled error checking. (See §4.2.3 for more details on error checking.) This returns a `TeamHandle` immutable object that encodes the previous team before entering the new context. When exiting that context, the handle is passed to `unsplit`, restoring the previous team context.

As with any lexical construct, such as the **`synchronized`** statement of Java, the compiler must ensure that the previous context is restored even when exiting the construct through an abrupt termination. An *abrupt termination* is a termination that exits a statement at any point other then its syntactic exit point and includes return statements and thrown exceptions. The Titanium compiler instruments abrupt terminations to restore the previous team context, if necessary.

Finally, when entering a new team context, the Titanium runtime sets the current GASNet team on each thread accordingly and updates a handful of variables to reflect the properties of the new team. As a result, there is very low overhead to switching team contexts, satisfying the performance goal of §3.1. In order to execute a collective, the Titanium runtime passes the current GASNet team to the GASNet collectives library, resulting in the desired behavior that collectives only execute over the current team.

Chapter 4

Alignment of Collectives

Many scientific applications are written in a bulk-synchronous style that alternates between communication and computation phases, or between different phases of physical simulations such as the ocean and atmospheric models in a climate simulation. These applications frequently require all threads to synchronize and communicate together. Like other SPMD languages, Titanium provides *collective operations* to support this. The four primitive collective operations in Titanium are barriers, broadcasts, exchanges, and reductions. A *barrier* forces threads to wait until all threads have reached it. A *broadcast* is a one-to-all communication construct that sends a value from one thread to the others. An *exchange* is an all-to-all communication construct that copies one value from each thread to all threads. A *reduction* combines values from each thread into a single value on one thread or an all threads. More complicated collectives can be built using these primitives, including global view operations like those in the data parallel model

Collective operations introduce the possibility of deadlock if not all threads execute the same sequence of collectives. The collectives are *aligned* if all threads do execute the same sequence.

Most SPMD languages do not attempt to guarantee alignment of collectives. Some languages such as UPC have *named* collectives. These collectives take an integer value as an argument. When the collective executes, it compares the value on all threads and generates an error if they differ. However, different collective operations in a program can have the same value under this scheme. Even if each collective in a program has its own unique value, as soon as the collective is wrapped inside a function, the alignment scheme can be defeated. For example, a call to the following function acts like an unnamed barrier:

```
void barrier2() {
  upc_barrier 315415431;
}
```

More complicated and less malicious examples of this can occur in practice. A further flaw with named collectives is that they can result in late error messages: the actual program statement[1] that causes misalignment can be far from the affected collective and is not detected or reported to the user.

[1]In this chapter, we use the term *statement* to refer to both statements and expressions in a program.

Aiken and Gay introduced the concept of *structural correctness* to enforce alignment of collectives and developed a static analysis that determines whether or not a program is structurally correct [1, 39]. The following code is not structurally correct:

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

Titanium provides a stronger guarantee of *textually-aligned collectives*: not only do all threads execute the same number of collectives, they also execute the same *textual* sequence of collectives. In addition, all control-flow decisions affecting execution of collectives must match on all threads, avoiding the problem with named collectives above. Thus, both the above structurally incorrect code and the following structurally correct code are erroneous in Titanium:

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

The fact that Titanium collectives are textually aligned not only guarantees deadlock freedom but also that code immediately following two different barrier operations cannot execute simultaneously. In §7, we rely on this fact to define an efficient but precise concurrency analysis.

Titanium currently relies on a static type system to ensure textual alignment of collectives. We discuss this type system and its drawbacks in §4.1. We then introduce a new dynamic scheme for enforcing textual alignment, starting with global collectives and then extending it to collectives over teams.

4.1 The Single Type System

In order for collectives to be textually aligned, all expressions that control the execution of collectives must evaluate to coherent values on all threads. In Titanium, the *single type system* ensures that this is the case. A value is *single* if it is *replicated and coherent* across all threads, meaning that all threads must have a copy of the value, and the values must have some semantic equivalence. The entire set of rules for what values are coherent is described in the Titanium language reference [43] and is fairly complicated, so we describe only a subset of the rules here.

4.1.1 Single Values

For primitive types, the coherency rules are straightforward. Compile-time and runtime constants (such as the number of executing threads) are single, as well as expressions composed entirely of single values. Variables are single if they are annotated as such by the programmer. Such variables can only by assigned with single values.

The rules for method calls are more complex. In order for the result of a method call to be single, its return type must be declared as single, the object being dispatched on must be single for an instance method, and all parameters declared as single must be passed single arguments. These rules are illustrated below:

```
class Foo {
  int single bar(int single x, int y) { ... }
  static void baz() {
    Foo single a = ...;
    Foo b = ...;
    int single i = 1;
    int j = Ti.thisProc();
    a.bar(i, j); // return is single
    b.bar(i, j); // return is not single since b is not
    a.bar(j, i); // return is not single since x = j is not
  }
}
```

A non-array allocation results in a single object if the constructor call obeys the rules for method calls above. A field dereference is single if the referenced object is single and the field is declared as single. Finally, an array allocation results in a single array if the size of the array is single. An array also can be declared to hold only single values, implying that the elements of the array are coherent across all threads.

4.1.2 Control-Flow Restrictions

All control-flow decisions that can affect the execution of collective operations must only depend on single expressions. Collective operations can be buried beneath many layers of code, such as function calls, so the following definition is useful:

Definition 4.1.1. (Global Effects) A statement has *global effects* if it or any of its substatements is a primitive collective operation, a method call that a programmer has declared as global by qualifying it with the **sglobal** keyword[2], or an assignment to certain locations that are declared as single.

All branches, loops, and method calls that have global effects must only be controlled by single expressions. Exceptions have more complicated restrictions, and the associated rules can be found in the Titanium language reference [43].

4.1.3 Problems in the Single Type System

As hinted at above, the single type system rules are complicated. Feedback from Titanium users indicates that while they appreciate the fact that the type system prevents deadlock, the error mes-

[2]In the current Titanium language specification, the **sglobal** keyword has been deprecated in favor of the **single** keyword. In this thesis, however, we will use **sglobal** to prevent confusion with non-global methods that have a single return or single arguments.

sages can be confusing at times due to the conservative nature of the analysis.

The type system requires the programmer to annotate many variables with the **single** keyword. Since a single variable can only be assigned an expression composed of other single variables, it may be necessary to propagate these annotations throughout a program. This can be quite burdensome, especially for quick prototyping of small pieces of code.

There are additional problems in the type system, such as its handling of arrays and casts to single. Most importantly, we have not yet found a clean way to extend the type system to allow collectives over thread teams. We discuss all three issues in more detail below.

4.1.3.1 Arrays and Polymorphism of Single

The Titanium type system treats the **single** qualifier on method parameters, method returns, and instance fields polymorphically, depending on use. A method with the signature

```
  static int single foo(int single x);
```

may be applied to both single and non-single arguments, returning **int single** for the former and **int** for the latter. Similarly, given the type definition

```
class Int {
  int single val;
  int single intValue() {
    return val;
  }
}
```

and variables `x` of type `Int` **single** and `y` of type `Int`, `x.val` and `x.value()` return **int single** while `y.val` and `y.value()` return **int**.

When it comes to array types, however, the **single** qualifier is only polymorphic at the top-level. As explained above, an array can be declared as **single** at the top level, as in **int**[] **single**. This indicates that it has the same number of elements on each thread. An array can also be declared as **single** at the element level, such as **int single**[] **single**, implying that the elements are coherent across all threads. Thus the method

```
  static int single bar(int[] single x) {
    x[0] = Ti.thisProc(); // set to non-single value
    return x.length; // return single value
  }
```

can be applied to arrays of type **int**[] and **int**[] **single**, returning an **int single** in the latter case. The method cannot, however, be applied to **int single**[] **single** arrays, as the assignment would be illegal. On the other hand, the method

```
  static int single baz(int single[] single x) {
    Baz.val = x[0]; // set a single variable to a single value
    return x[1]; // return single value
  }
```

can only be applied to arrays of type **int single**[] **single**. If Baz.val is a static variable declared as **int single**, the assignment would be invalid if the input array were not **single** at the element level.

This makes it impossible to use the **single** qualifier to specify coherence of array-based data structures. Consider two definitions of integer vectors.

```
class IntVector1 {
  int[] single values;
  int single length() {
    return values.length; // OK
  }
  int single elementAt(int single i) {
    return values[i]; // type error -- values[i] is non-single
  }
}

class IntVector2 {
  int single[] single values;
  int single length() {
    return values.length; // OK
  }
  int single elementAt(int single i) {
    return values[i]; // OK
  }
}
```

The first definition, IntVector1, can be used polymorphically, but it contains a type error since the **single** qualifier at the top-level of an array only specifies that the array has the same length on all threads and says nothing of their elements. The second definition, IntVector2 is correctly typed, but cannot be used polymorphically.

This limitation is particularly relevant when it comes to the String class. The contents of a String **single** are contained in an array of type **char**[] **single**. Since this array is single only at the top-level, its elements are not guaranteed to be coherent across all threads.

4.1.3.2 Casts to Single

The Titanium type system allows arbitrary casts to single, which go unchecked at runtime. This is used for convenience by some users to circumvent the type system, but it is also used to get around the limitations discussed in §4.1.3.1. For example, a polymorphic integer parsing function could be written as follows.

```
static int single parseInt(String single s) {
  int single result = 0;
  for (int single i = 0; i < s.length(); i++) {
    result = 10 * result +
      (int single) Character.getNumericValue(s.charAt(i));
  }
  return result;
}
```

This procedure requires a cast to **single** in order to function as desired. The cast is unsafe, however, since the contents of a `String` **single** are not guaranteed to be coherent.

Unfortunately, in a case like this, it is unclear how to check the cast at runtime. This is because `parseInt()` is polymorphic – it may be called by all threads, in which case the cast can be checked using communication, but it may also be called by a subset of the threads, in which case communication cannot be done.

4.1.3.3 Team Collectives

The most significant drawback to the single type system is that there is no obvious way to extend it to enforce alignment of collectives over thread teams. Consider the following code that uses the partition construct introduced in §3.

```
static int single x;
static Team single team;
static single void bar() {
  partition(team) {
    { setX(1); }
    { setX(2); }
  }
  if (x == 1) {
    Ti.barrier(); // misaligned barrier
  }
}
static single void setX(int single y) {
  x = y;
}
```

Since **single** is used to ensure alignment of collective operations, it only implies coherence across all threads of a given team. This can cause problems if a single variable is updated within a partition or teamsplit statement. In the code above, threads in different teams assign different values to the single variable `x`. Since all threads in a team assign the same value to `x`, the code is correctly typed. However, it will deadlock, since the barrier after the partition is executed by all the threads, which now have incoherent values of `x`.

There appears to be no easy solution to the fact that teams are lexically scoped, while single variables may have global scope. We can attempt to restrict single variables from being assigned in a different team context from which they were defined, but that would also restrict methods such as `setX` that assign to static single variables from being called in the context of any team other than the global one. This would be potentially confusing to programmers, and it is less clear how single instance variables would be handled.

4.2 Dynamic Alignment

Given the problems in the single type system described in §4.1.3, we present an alternative, dynamic scheme for enforcing textual alignment of collectives. Such a scheme provides advantages over not checking alignment at all. On many supercomputing and distributed computing environments, users are charged according to time used. An application run that deadlocks can waste many compute hours before the problem is noticed, resulting in actual cost to the user. While deadlock detection schemes can alleviate this problem, they only detect erroneous symptoms rather than the cause. In particular, misalignment earlier in a program may result in a deadlock much later, and deadlock detection would not report the actual source of the alignment error. Our dynamic alignment scheme, on the other hand, does so, preventing deadlocks from occurring in the first place.

4.2.1 Alignment Rules

The basic conditions that guarantee textual alignment of collectives are as follows[3]:

1. If any branch of a conditional has global effects, then all threads must take the same branch.
2. If the body/test of a loop has global effects, then all threads must execute the same number of iterations.
3. If a method call has global effects, then the dynamic dispatch target of the call must be the same on all threads.
4. The source thread in a broadcast expression and target thread in a reduction must evaluate to the same value in each thread.

All four conditions above are enforced by the single type system.

Multiple alignment schemes can be defined based on different definitions of global effects.

Definition 4.2.1. (Strict Alignment) In *strict alignment*, a statement has global effects if it or any of its substatements is a primitive collection operation or calls a method declared as **`sglobal`**.

Strict alignment uses a similar definition of global effects as the single type system. The only difference is that it does not restrict assignments to locations declared as single, since such declarations are not present in dynamic alignment. A weaker scheme is possible as follows:

Definition 4.2.2. (Weak Alignment) In *weak alignment*, a statement has global effects if it or any of its substatements *executes* a primitive collection operation at runtime.

To illustrate the difference between strict and weak alignment, consider the following code that is legal under weak alignment but prohibited under strict alignment:

```
if (Ti.thisProc() % 2 == 0) // even threads
  if (Ti.thisProc() % 2 == 1) // odd threads
    Ti.barrier(); // never reachable
```

[3]We omit discussion of exceptions here, as the rules are essentially the same as in the single type system.

Under weak alignment, the code above never executes the barrier, so it does not have global effects and not all threads must take the same branch of the outer conditional. Under strict alignment, however, the *then* branch does have global effects since one of its substatements is a barrier, so all threads must take the branch.

An important feature of strict alignment is that its rules are static: it is possible to determine which statements have global effects at compile-time. The weak alignment rules, on the other hand, are partially dynamic. For some statements, it can be statically determined that they never execute primitive collectives. For others, however, it can only be determined at runtime whether or not they do so. As a result, we believe that strict alignment is preferable both for compiler analysis purposes and for programmer reasoning.

4.2.2 Dynamic Enforcement

The alignment rules are enforced dynamically by tracking those conditionals, loops, and method calls that have global effects. (For the purposes of this section, global effects are according to the strict definition above.). The Titanium compiler has an inference system that statically determines which statements have global effects.

At program startup, each thread creates an empty list that records its execution history. In addition, a hash of this list is maintained, initially set to some value h_0. The following operations update the list, with the hash updated accordingly:

- On a non-static dispatch to a method that has global effects, an entry is added to the list with the method that is the dynamic dispatch target.
- On a branch of a conditional that has global effects, an entry is added recording the branch taken.
- On each iteration of a loop that has global effects, an entry is added recording that a loop iteration occurred.
- On reaching a broadcast or reduction operation, an entry is added with the value of the source thread specified by the broadcast or the target thread specified by the reduction.

When performing a primitive collective, the hashes for all threads are first compared. This can be done by using a comparison tree, or simply by broadcasting the hash value from a single thread and comparing it to the value on each other thread. Our implementation currently uses the latter. If the hash values match, the collective is executed. If any two hashes differ, however, then execution halts, and the corresponding histories are used to generate an appropriate error message. For performance reasons, the execution history list can be eliminated or reduced in size, at the cost of poorer error messages.

The above procedure is sufficient for enforcing strict alignment. Weak alignment, on the other hand, only guarantees alignment of statements that execute primitive collectives at runtime. Thus, if a statement never executes a primitive collective, the execution history and hash must be restored

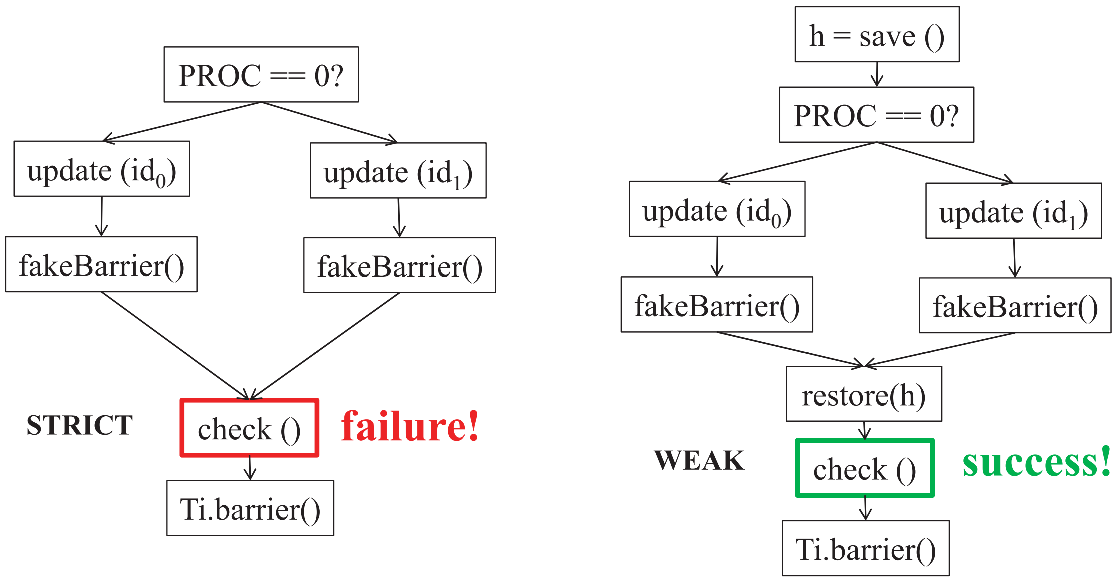


Figure 4.1: Difference in execution between strict and weak alignment.

to their previous values once the statement completes. This necessitates saving the old history and hash state before making any of the above changes.

Figure 4.1 illustrates the execution of the following code under strict and weak alignment:

```
if (Ti.thisProc() == 0) {
  fakeBarrier();
} else {
  fakeBarrier();
}
Ti.barrier();
```

Here, `fakeBarrier` is a method declared as **`sglobal`** but that does not execute any primitive collective operations. As such, the code should fail under strict alignment but succeed under weak alignment.

4.2.2.1 Optimizations

It is possible to reduce the number of history updates and checks if they can be proven redundant. If two history updates always occur in sequence with no intervening collectives, then they can be combined into a single update. This may occur in nested conditionals or conditionals inside loops. Similarly, two history checks can be redundant if no updates occur between them.

Another optimization is a hybrid static/dynamic analysis that would apply a static analysis to determine which branches, loops, and method calls can be proven to depend only on single values and then eliminate their associated updates. This would also make it much more likely for consecutive collectives not to have any updates between them, allowing their associated history checks to be removed.

4.2.2.2 Program Coverage

As is usually the case with dynamic analysis, the above enforcement scheme does not provide full program coverage, which is a drawback compared to the static type system. In particular, it does not check alignment of code that is not reached at runtime, such as the following:

```
if (<some rare condition>)
  if (Ti.thisProc() == 0)
    Ti.barrier();
```

An error is only generated if the rare condition is taken. Similarly, if the rare condition was replaced by an expression dependent on the number of threads, such as `Ti.numProcs()`> 100, then an error would only occur if the program was run with the required number of threads to trigger the condition.

Weak alignment provides somewhat less coverage than strict alignment. Consider the above code with the two conditions switched. Now, strict alignment would generate an error since the threads are not aligned with respect to the outer conditional, which has global effects. Weak alignment, on the other hand, will only find an error if the rare condition is met, since no primitive collective operation is executed otherwise.

4.2.3 Implementation

We have implemented the two dynamic enforcement schemes in the Titanium compiler. The compiler instruments each program to perform the required tracking and checking[4]. We do not apply the optimizations described in §4.2.2, since our experimental results in §8.2.2 show them to be unnecessary.

For both strict and weak alignment, the compiler provides a default mode where only a hash is kept corresponding to execution history as well as a debugging mode that maintains the execution history list. The default mode requires less memory and fewer operations at runtime than the debugging mode, so it could potentially be more efficient. The debugging mode only stores execution history between successive primitive collective operations, since successful completion of a collective implies that it and all statements preceding it are properly aligned.

The Titanium compiler provides an escape hatch for when dynamic error checking adversely affects performance. Users generally switch to a less-safe, higher-performance mode that elides error checking such as **`null`**-pointer and array-bounds checks for production runs, under the assumption that such errors have already been caught while debugging a program. This escape hatch can also be used if dynamic alignment checking proves to be too expensive, as the compiler can remove those checks as well.

[4]For weak alignment, we could instead examine the program stack on each thread. However, this would require a far more complicated implementation, as stack layout depends on the target machine and the C compiler.

4.3 Alignment of Team Collectives

We now turn our attention to alignment of collectives over thread teams. We extend the global dynamic alignment scheme of §4.2 to handle team collectives as well. In addition, the new scheme ensures that a program makes proper use of the partition and teamsplit constructs introduced in §3.

4.3.1 Alignment Rules

Titanium requires that collectives be textually aligned on all threads that participate in a collective. In order to guarantee this, we require a definition of team effects rather than global effects. As with global effects, the exact definition of team effects depends on the alignment scheme used.

Definition 4.3.1. (Strict Alignment Team Effects) In strict alignment, a statement has *effects on team* t if it or any of its substatements is a primitive collection operation on team t or calls a method declared as **`sglobal`** in the context of t.

Definition 4.3.2. (Weak Alignment Team Effects) In weak alignment, a statement has *effects on team* t if it or any of its substatements *executes* a primitive collection operation on team t at runtime.

We proceed to define an enforcement scheme that works for both strict and weak alignment. The basic conditions that guarantee collective alignment consist of the following global rules, which are an extension of the rules in §4.2.1:

1. If any branch of a conditional has effects on team t, then all threads in t must take the same branch.
2. If the body/test of a loop has effects on team t, then all threads in t must execute the same number of iterations.
3. If a method call has effects on team t, then the dynamic dispatch target of the call must be the same on all threads in t.
4. The source thread in a broadcast and target thread in a reduction on team t must evaluate to the same value on each thread in t.
5. `Team` objects passed to a partition or teamsplit on team t must have consistent immediate subteams across all threads in t.
6. The level argument passed to a superset statement or collective that targets team t must evaluate to the same value on all threads in t.

In addition, the following local rules must be satisfied:

7. `Team` objects passed to a partition or teamsplit must match the current team at the top level.
8. A superset operation with a level argument i must be enclosed by i teamsplit statements, with no intervening partition statements.

4.3.2 Alignment Enforcement

The basic idea behind the dynamic enforcement system in §4.2.2 and the extension here is that each thread tracks all control-flow decisions that potentially affect alignment of a collective. Prior to executing a collective, the threads cooperate to perform a global check to ensure that they are all aligned. Since this check runs before each collective, misaligned collectives are never executed, as they are detected in the preceding check. A failed check results in the program terminating with an error message describing the sequence of control-flow decisions that resulted in the misalignment. An implicit barrier with a corresponding alignment check occurs at program end, catching any unmatched collectives.

The first four alignment rules are enforced by inserting an entry in a per-thread tracking list recording the decision made when executing an affected program expression or statement. We also maintain a running summary hash, as described in §4.2.2. Since partition and teamsplit statements are collectives themselves, rules 5 and 7 can be checked directly when entering such statements. We come back to rules 6 and 8 later.

In order to extend dynamic enforcement to work on all teams, we now keep separate track of control-flow decisions that may affect alignment of collectives on different teams. To simplify the discussion, we ignore the existence of superset operations until later. Then it is sufficient for both strict and weak alignment to record control-flow decisions in the tracking list for the current team. Upon encountering a collective operation, the tracking list is checked for consistency among only the threads in the current team. A final check must be made at the end of a partition or teamsplit to ensure that no unmatched collectives exist within such a statement.

As a concrete example, consider the following code:

```
1 if (a) Ti.barrier();
2 teamsplit(u) {
3   if (b) Ti.barrier();
4 }
5 if (c) Ti.barrier();
```

Let t be the current team outside the teamsplit. When the conditional on line 1 is executed, each thread in team t records the branch taken in the tracking list for t. Those threads that take the *then* branch await a check before performing the barrier. If other threads do not take this branch, they perform a check before the teamsplit, resulting in the error being detected. Upon encountering the teamsplit and executing a successful check, the threads in t ensure that team u is equivalent to t at the top level and that the immediate subteams of u are the same on all threads. If this is the case, then each thread's corresponding subteam in u becomes the new current team on that thread, and control-flow decisions within the teamsplit are now recorded in the tracking list for `u.myChildTeam()`. Thus, it is perfectly valid for one subteam of u to execute the barrier on line 3 while other subteams skip it. Let v refer to the first subteam of u. Then all threads in v record the branch taken, and if some threads in v take the *then* branch, they will await a check before performing the barrier. If other threads do not take the branch, then they will perform a check at the end of the teamsplit, resulting in detection of the error. Finally, upon leaving the teamsplit, t

once again becomes the currently executing team, so that alignment of the barrier on line 5 will be checked with respect to team t.

To ensure that alignment is checked with respect to the proper team, it is necessary to ensure that alignment is consistent in the current team before entering a new team context. The checks prior to teamsplit and partition and at their end ensure that this is the case.

4.3.2.1 Superset Operations

Superset operations complicate alignment checking since they take in a level parameter that may not be known at compile-team. Consider the following code:

```
if (d) {
  int n = ...; // not a compile-time constant
  Ti.barrier(n);
  Ti.barrier(n+1);
}
```

Prior to entering the branch, a thread may not be able to determine at what levels the barriers execute and thus which teams' tracking lists need to be updated. In order to handle this, an orphan tracking list must be maintained that keeps track of control-flow decisions that may affect alignment of a superset operation. Then when such an operation is encountered, the orphan list's contents are copied into that of the teams affected by the operation.

A superset operation is defined to have team effects on all teams between the current team and the target team[5]. Thus, in checking a superset operation, alignment must first be checked at the current team level and all intermediate levels up to the target level, in order, to ensure that alignment is satisfied at every level. Then rules 6 and 8 in §4.3.1 are checked before performing the superset operation.

4.3.3 Implementation

We have implemented team extensions to dynamic enforcement in the context of weak alignment. As alluded to in §4.3.2, separate tracking lists are recorded for each team, along with separate summary hashes. Upon encountering a partition or teamsplit, team arguments are compared on all threads in the enclosing team, ensuring that the arguments match. Prior to executing any collective operation on a particular team, the runtime compares hashes for that team only, since the partition and teamsplit checks guarantee that all threads in that team are in the same team context.

As with global alignment checks, we allow a user to turn off team alignment checks in order to eliminate their overhead. This includes both the specific partition and teamsplit checks as well as those for general collectives. We take advantage of this facility in the application case studies in §8.1.

[5]In strict alignment, a superset operation that is not executed at runtime is defined to have effects only on the current team, since the target team is unknown.

Chapter 5

Analysis Background

We now turn our attention to the analysis of recursive single program, multiple data (RSPMD) programs. Prior work has demonstrated that the flat SPMD model enables many simple yet precise analyses, including data locality analysis [65], data sharing analysis [66], concurrency analysis [26, 78, 99, 45, 111, 52], and pointer analysis [54]. We would like to show that RSPMD programs are similarly amenable to analysis.

We begin by describing the machine model used in our analyses, followed by an overview of pointer and concurrency analysis and their applications to further program analysis and optimization. We then demonstrate how team information can improve analysis of RSPMD programs and discuss a possible abstract representation of this information for use in pointer and concurrency analysis.

5.1 Machine Model

Consider a set of machines arranged in an arbitrary tree hierarchy with the machines as leaves, such as that of Figure 5.1. A *machine* constitutes a single computational element in the system, and we assume a one-to-one correspondence between machines and threads in a program. Each machine has a corresponding *machine number*. The *depth* d of the hierarchy is the number of levels it contains. The *distance* between machines is equal to the number of levels from the *bottom* of the hierarchy to that containing their least common ancestor, with $d - 1$ denoting the top level. The distance is thus $d - 1 - n$, where n is the level number of the least common ancestor. A pointer on a machine m has a corresponding *width*, and it can only refer to locations on machines whose distance from m is less than or equal to the pointer's width minus one. Thus, width is in the range $[1, d]$. (We use one-indexing for width, in which lower numbers refer to lower levels in the machine hierarchy, to distinguish it from level indices in a hierarchy, in which lower numbers refer to higher levels.) Since each pointer has a width and the set of possible widths corresponds to the number of levels in an arbitrary machine hierarchy, the model we define here is an instance of the hierarchical partitioned global address space (HPGAS) memory model.

The distance function defined above is an *ultrametric*, since it satisfies the following properties

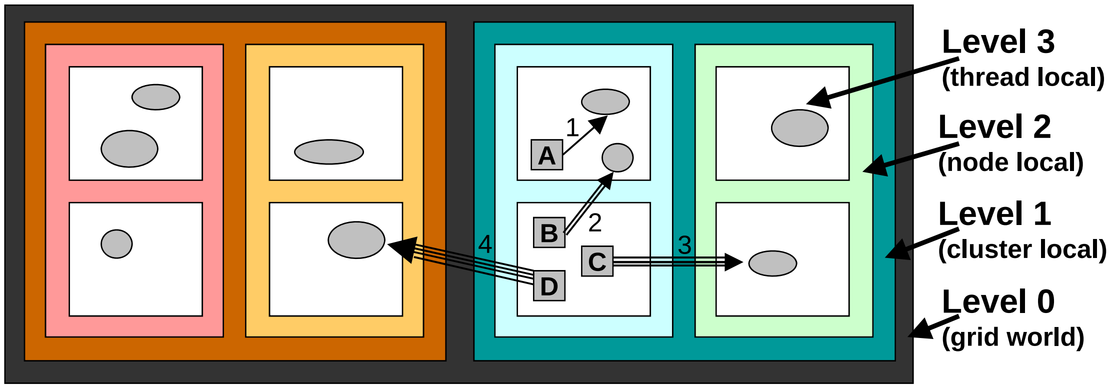


Figure 5.1: A possible machine hierarchy with four levels. The width of arrows and their labels indicate the hierarchy distance between the endpoints, and the level numbers at right correspond to those in the machine team.

for all machines x, y, z:

$$\begin{aligned} d(x,y) &> 0 \quad \text{if } x \neq y \\ d(x,y) &= 0 \quad \text{if } x = y \\ d(x,y) &= d(y,x) \\ d(x,y) &\leq d(x,z) + d(y,z) \\ d(x,y) &\leq \max(d(x,z), d(y,z)) \end{aligned}$$

A *k-hierarchically well-separated tree* (k-HST) is a special case of an ultrametric. In such a tree, the lengths of each path segment from the root to any leaf decrease by a factor of k in each step. Fakcharoenphol, Rao, and Talwar showed that any graph metric can be approximated by a 2-HST within a factor of $\mathrm{O}(\log n)$, where n is the number of nodes in the graph [35]. Though the distance function above is a 1-HST, the pointer analysis can be easily modified to use 2-HST functions, allowing arbitrary metric machine topologies to be well-approximated by the analysis.

5.2 Pointer Analysis

Pointer analysis, or *points-to analysis*, statically determines the set of objects that may be referenced by each pointer in a program. Pointer analysis was first described and implemented for the C programming language by Emami [34] and Andersen [6]. Here, we provide an overview of context-insensitive, flow-insensitive pointer analysis for object-oriented languages such as Java and Titanium.

Pointer analysis is an example of *abstract interpretation*, where the execution of a program is approximated statically. In the case of pointer analysis, each allocation site corresponds to an *abstract location*, which represents all objects allocated at that site at runtime. Each allocation

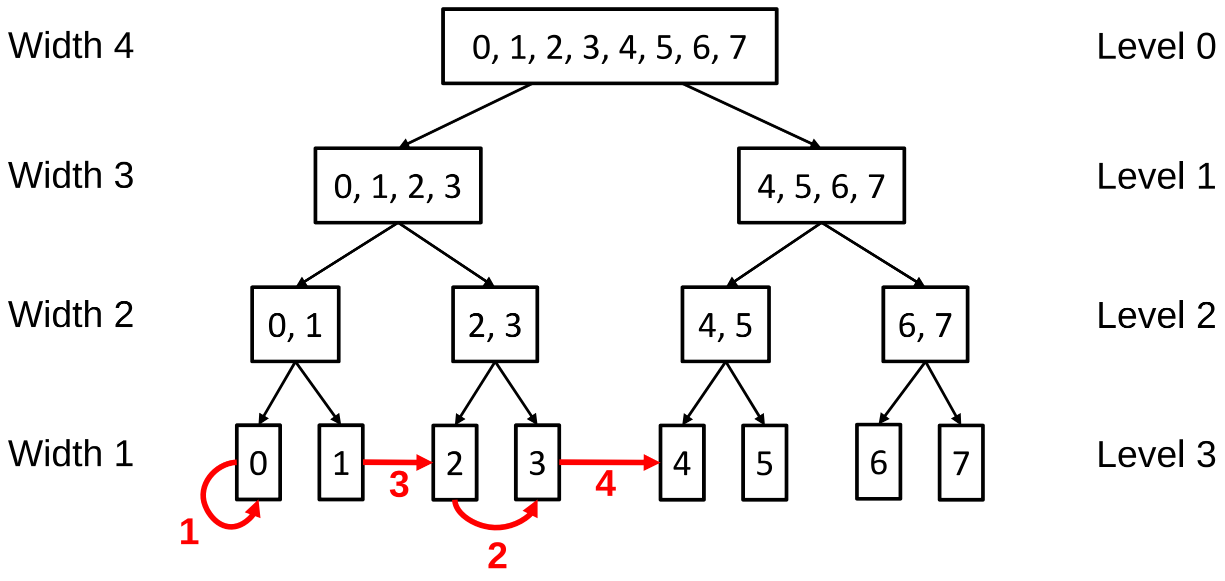


Figure 5.2: Example of pointer width with respect to the machine hierarchy team.

site only constructs objects of a single type[1], and its corresponding abstract location has the fields specified by that type. Every variable in the program and field of each abstract location has a *points-to* set consisting of abstract locations that may be referenced at runtime by that variable or field. The analysis iterates over the whole program, updating points-to sets according to statements and expressions in the program. For example, consider the assignment

```
v = x.f;
```

The analysis examines the points-to set of `x`, and for each abstract location in that set, it examines the points-to set for the field `f`, collecting the resulting abstract locations. These locations then are copied into the points-to set of `v`. The analysis continues to iterate over the program, interpreting each statement and expression, until a fixed point is reached.

We have previously described a pointer analysis for Titanium that takes into account machine hierarchy [54]. As described in §5.1, each pointer has a width in the range $[1, d]$ where d is the depth of the machine hierarchy, restricting the set of threads on which the referenced data can be located. Figure 5.2 shows examples of pointer width with respect to a sample machine hierarchy team.

In the hierarchical pointer analysis, abstract locations have both a corresponding allocation site and a width. For each allocation site, d abstract locations are created, each with a separate width. When an abstract location escapes its creating thread, either through a collective operation such as a broadcast or through a dereference, the result is an abstract location with the same allocation site but a greater width. As a concrete example, consider the following statement:

```
w = broadcast x from 0;
```

[1]We ignore Java reflection for simplicity.

This statement assigns thread 0's value of `x` to `w` on all threads. Suppose that in the pointer analysis, the points-to set of `x` contains the lone abstract location $(l, 1)$, meaning the abstract location with width 1 corresponding to allocation site l. Then the broadcast results in the abstract location (l, d), which is added to the points-to set of `w`. The resulting width is d since the broadcast is a global operation over all threads.

5.2.1 Applications

Many client analyses and optimizations can take advantage of the information computed by pointer analysis. Three examples are locality analysis, sharing analysis, and race detection.

5.2.1.1 Locality Analysis

In partitioned global address space (PGAS) languages such as Titanium, a pointer can reference data located on any thread, even if the source and target of a pointer do not share the same physical memory space. A pointer has a particular width that specifies where the referenced data may be located, with greater widths allowing reference to data on more distant threads. In Titanium, pointers have maximal width by default and must be qualified as **`local`** in order to specify a smaller width. Unfortunately, pointers with greater width can be more expensive to dereference than pointers with smaller width, even when the target location is the same, since a runtime check of where the data is located may be required in the former case. Liblit and Aiken demonstrated up to a 56% improvement in application running time with an automated analysis to infer local qualifiers [65]. This illustrates the need for a *locality analysis* to compute locality information at compile time.

Liblit and Aiken's analysis uses a constraint solver in order to compute locality information. Unlike pointer analysis, their analysis does not distinguish between allocation sites, so pointer analysis should be able to produce more precise results. Computing locality information from points-to sets is trivial in a hierarchical pointer analysis: the width of a variable is the maximum width of any of the abstract locations in its points-to set.

5.2.1.2 Sharing Analysis

In parallel programs, *sharing analysis*, also called *escape analysis*, is relevant to many applications. Sharing analysis statically determines what data may be shared between multiple threads. If an object is provably private to a single thread, a compiler can safely allocate it in private memory, such as the thread's stack or scratchpad memory. Synchronization can also be eliminated for private objects. This is especially important in library code that tends to be over-synchronized for safety. For example, the `java.util.Vector` is synchronized, so an application that uses thread-private instances of the class can execute many spurious synchronizations. Bogda and Hölzle demonstrated up to a 36% improvement in benchmark performance by automatically eliminating synchronization on private objects in Java applications [16]. Others have also used sharing analysis to eliminate synchronization [3] or to enforce a sequentially-consistent memory model [64, 95, 96, 60, 91, 66, 51].

Hierarchical pointer analysis can be used for sharing analysis. An allocation site is private if no points-to set contains an abstract location corresponding to that site with width greater than 1. A variable is private if all abstract locations in its points-to set correspond to private allocation sites.

5.2.1.3 Race Detection

A *race condition* occurs when two memory accesses can occur concurrently on different threads, they can be to the same memory location, and at least one of them is a write [76]. Race conditions can result in erroneous program behavior, so static detection of races is an important problem in program analysis.

Pointer analysis is a key component of race detection, since it can determine whether or not two memory accesses may be to the same location. Variables are said to be *aliased* if they may reference the same location. In hierarchical pointer analysis two variables `x` and `y` are *aliased across threads* if:

- The points-to set of `x` contains an abstract location (l, a) and that of `y` contains a location (l, b) that correspond the the same allocation site.
- Either a or b is greater than 1, meaning that at least one the referenced locations may not be thread-local and thus may overlap with the other.

As an example, consider the following code:

```
1 Foo w = new Foo();
2 Foo z = broadcast w from 0;
3 w.f = 4;
4 return z.f;
```

The points-to set of `w` contains the abstract location $(1, 1)$, so that of `z` contains the location $(1, d)$, where d is the depth of the machine hierarchy. Thus `w` and `z` are aliased across threads, and `z` is aliased with itself across threads. On the other hand, `w` does not alias itself across threads. As a result, the write on line 3 does not by itself constitute a race condition, since the writes are to thread-local locations on all threads. However, the combination of that write and the read on line 4 is a race condition, since other threads read the location in line 4 that is concurrently written in line 3.

5.3 Concurrency Analysis

Information concerning which statements and expressions may run concurrently is important in many analyses and optimizations. *Concurrency analysis* statically computes this information, analyzing the synchronization structure of a parallel program. We previously described a precise and efficient concurrency analysis for SPMD programs [52]. This analysis traverses a graph representation of a program, and two expressions are never concurrent if all paths between them contain a global synchronization operation. We review the analysis in more detail in §7.1.

As an example of concurrency analysis, consider the following code that makes use of a barrier operation:

```
1 Foo w = new Foo();
2 Foo z = broadcast w from 0;
3 w.f = 4;
4 Ti.barrier();
5 return z.f;
```

A *barrier* operation requires all threads to reach it before any can proceed. In the above code, the only path from the write on line 3 to the read on line 5 contains a global barrier synchronization. As a result, they never execute concurrently, and the code does not contain a race condition.

Concurrency information is useful for analyses and optimizations besides race detection. For example, it can be used in synchronization elimination, particularly in eliminating redundant barrier operations [26, 78, 99]. In addition, concurrency information can be helpful when providing a sequentially-consistent memory model, as we demonstrate next.

5.4 Sequential Consistency

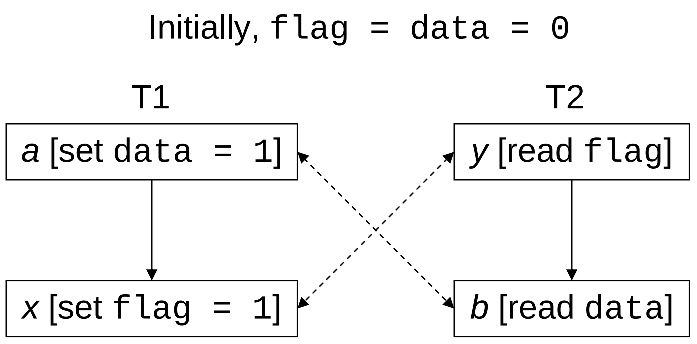
For a sequential program, compiler and hardware transformations must not violate data dependencies: the order of all pairs of conflicting accesses must be preserved. Two memory accesses *conflict* if they access the same memory location and at least one of them is a write. The execution model for parallel programs is 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 memory locations that are shared between them. A memory consistency model defines the memory semantics and restricts the possible execution order of memory operations.

Titanium's memory consistency model is defined in the language specification [43]. Here are some informal properties of the Titanium model.

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

2. **Globally consistent at synchronization events:** At a global synchronization event such as a barrier, all threads must agree on the values of all the variables. At a team synchronization event such as a team barrier, all threads in the team must agree on the values of all variables. Finally, upon entry to a critical section, a thread must see all updates made prior to and within that critical section by threads that have previously entered it.

Henceforth, we will refer to the Titanium memory consistency model as the *relaxed model*.



y sees effect of x	b sees effect of a	possible sequential order
yes	yes	$a \Rightarrow x \Rightarrow y \Rightarrow b$
yes	no	none
no	yes	$a \Rightarrow y \Rightarrow b \Rightarrow x$
no	no	$y \Rightarrow b \Rightarrow a \Rightarrow x$

Figure 5.3: A cycle consisting of four accesses in two threads. The solid edges correspond to order in the execution stream of each thread, and the dashed edges are conflicts. Of the four possible results of thread 1 visible to thread 2, the second is illegal since it does not correspond to an overall execution sequence in which operations are not reordered within a thread.

A simpler memory model, *sequential consistency*, is the most intuitive for the programmer. The sequential consistency model states that a parallel execution must behave as if it were an interleaving of the serial executions by individual threads, with each individual execution sequence preserving the program order [61]. Figure 5.3 shows a simple set of operations on two threads, and for each possible execution result, whether or not a sequential interleaving exists.

An easy way to enforce sequential consistency is to insert memory barriers after each access to a memory location that may be shared. A *memory barrier* or *fence* forces all previous memory operations to complete before execution can proceed, preventing optimizations such as prefetching and code motion and resulting in an unacceptable performance penalty. Various techniques, such as *cycle detection* [91, 60], have been proposed to minimize the number of barriers, or *delay set*, required to enforce sequential consistency.

Computing the minimal delay set for an arbitrary parallel program is an intractable NP-hard problem [91, 60]. Krishnamurthy and Yelick proposed a polynomial time algorithm based on *cycle detection* for analyzing SPMD programs [60] such as Titanium. The analysis uses a graph where the nodes represent shared memory accesses. There are two types of edges in the graph: *program edges* and *conflict edges*. Program edges reflect the program order: there is a directed program edge from u to v if u can execute before v. Conflict edges are undirected edges between accesses that conflict: there is a conflict edge between u and v if u and v can access the same memory location and at least one of them is a write.

The goal of cycle detection is to check each program edge to see if it needs a fence to enforce its order. Given the program edge (u, v), if there is no local dependency between u and v, v could execute before u. If this reordering is observable by another thread, then sequential consistency is violated. In that case, a fence must be inserted between u and v to ensure that u always executes before v. Figure 5.3 gives one example of this. There is no local dependency on T1, but if the two writes on T1 were reordered, then the following execution order would be possible: $x \Rightarrow y \Rightarrow b \Rightarrow a$. This results in (y, b) reading the values $(1, 0)$, which means that the reordering on T1 is observable on T2. A fence must be placed between a and x to prevent such reordering.

Kirshnamurthy and Yelick [60] showed that given a program edge (u, v), if there is a path from v to u where the first and last edge are conflict edges, and the intermediate edges are program edges, then the program edge (u, v) belongs to the minimal delay set and a fence must be placed between u and v to prevent reordering. The path together with the program edge (u, v) forms a *critical cycle*.

Concurrency information can be used to reduce the set of conflict edges for a program. Suppose two memory accesses a and b conflict. We show that if a and b can never run concurrently within the relaxed memory model, it is possible to remove the resulting conflict edge since it can never take part in a cycle that violates sequential consistency.

Theorem 5.4.1. *Let* a *and* b *be two memory accesses in a program, and* C *a cycle containing the conflict edge* (a,b). *If* a *and* b *cannot run concurrently, then reordering* a *with another access does not violate sequential consistency with respect to the accesses in* C *in any execution of the program, as long as accesses are not moved across synchronization points.*

Proof. We prove this for a cycle consisting of four accesses in two threads where a is the first access in thread 1 and b is the second access in thread 2, as in figure 5.3 (the proof can be generalized to arbitrary cycles). Let x and y be the other two conflicting accesses in C, in thread 1 and 2 respectively. Consider an arbitrary execution in which the accesses in C occur. Since a and b cannot run concurrently, either a must complete before b or b must complete before a.
Case 1: a *occurs before* b. Sequential consistency can only be violated if y sees the effect of x, but b does not see the effect of a. In all other cases, execution corresponds to a valid sequentially consistent ordering, as shown in the table in figure 5.3. But since a occurs before b, b always sees the effect of a, so sequential consistency is preserved regardless of the order of a and x.
Case 2: b *occurs before* a. In order to enforce that b occur before a, there must be a synchronization point between b and a in the execution stream of each thread. Since accesses aren't moved across such points, y must occur before it and x must occur after it. This means that y must complete before x and therefore does not see its effect. Since y does not see the effect of x and b does not see the effect of a, the execution is sequentially consistent independent of the order of a and x. □

Since a conflict edge in which the two accesses may be concurrent is by definition a race condition, the above proof demonstrates an important property: *race-free code is sequentially consistent.* Programmers rarely write code that intentionally has race conditions. One exception is spin locks to protect shared data, such as the following, which results in the critical cycle shown in Figure 5.3:

```
// thread 0 code
```

```
2 data = 1;
3 flag = 1;
4
5 // thread 1 code
6 while (flag == 0);
7 print(data);
```

Such code may execute incorrectly in a memory model that is not sequentially consistent. It also contains two races in the context of a relaxed memory model: the write to `data` on line 2 and read from it on line 7, and the write to `flag` on line 3 and read from it on line 6. In both cases, the writes and reads occur with no intervening synchronization operation, resulting in race conditions. These races correspond to the two conflict edges in the resulting critical cycle. Thus, enforcement of sequential consistency is tied directly to race detection.

5.5 Analysis over Teams

With the addition of teams, collective operations need not be global. As a result, widening abstract locations to the maximum width in pointer analysis may be overly imprecise. For example, a broadcast over a lower level in `Ti.defaultTeam()`, the team that matches the machine hierarchy, should produce an abstract location with width smaller than d. Thus, team-awareness can improve locality analysis.

In addition to width with respect to the machine hierarchy, it would be useful in pointer analysis to compute widths with respect to arbitrary team hierarchies. Similarly, concurrency analysis can compute team-level concurrency information in addition to the global information it computes now. As an example, consider the following code:

```
1  Foo y;
2  teamsplit(t1) {
3    Foo x = new Foo();
4    y = broadcast x from 0;
5    x.f = 4;
6  }
7  ...
8  teamsplit(t2) {
9    Ti.barrier();
10   return y.f;
11 }
```

If `t1` and `t2` reference the same concrete team at runtime, say T_a, then the write on line 5 does not conflict with the read on line 10. This is due to the fact that the threads that read the location `y.f` and the thread that writes to it belong to the same child team of T_a, so `x` and `y` are aliased across the child team of T_a but not globally. However, all threads in that child team synchronize on the barrier in line 9, so the two accesses are non-concurrent in the child team of T_a, and no race condition exists. On the other hand, if `t1` and `t2` do not refer to the same team, then a race condition may result. Thus, knowledge of the set of possible active teams at each program location is essential to precise pointer and concurrency analysis.

Team information may be computed by a static analysis, as we propose in a companion report [48]. Alternatively, we can rely on user annotations in order to provide precise and accurate information about the teams in a program and their usage.

5.5.1 The Team Lattice

Once the set of teams in a program and their relationship to each other is determined, whether through analysis or annotations, a *team lattice* is constructed to represent this information. This lattice is used directly in hierarchical pointer analysis (§6) and indirectly in concurrency analysis (§7) to construct the lattices required there.

The team lattice has a single element for each unique team-hierarchy level in the program; if two levels in two team hierarchies are equal, they share a lattice element, even if their children differ. Edges in the team lattice represent parent-child relationships between teams. A new lattice element is added to represent thread-local operations, and an edge is added from that element to each team that has no children. A minimal $\perp$ element represents null operations, and an edge is added from it to thread local. Finally, the maximal element $\top$ corresponds to the global team. Figure 5.4 shows an example of a team lattice.

Each path from the top of the lattice to thread local represents a distinct team hierarchy. The height of the lattice is the maximum height of all team hierarchies plus one, and we use d_{lat} to denote this value.

Finally, each team t has an associated *locality*, which corresponds to the lowest level in the machine hierarchy that contains all threads in t: the locality of a team is the minimum pointer width that can reference all data located on a team's threads. This is not necessarily the lowest element of the machine team that is an ancestor of t, as the example below demonstrates.

5.5.2 Example

As a concrete example of a team lattice and team usage information, consider the following code that executes in the context of the global team:

```
1 Team a = Ti.defaultTeam();
2 Team b = new Team();
3 Team c;
4 b.splitTeam(2);
5 teamsplit(b) {
6   b.myChildTeam().splitTeamSharedMem(Ti.thisProc());
7   c = new Team();
8   c.splitTeam(2);
9 }
```

There are three team hierarchies in this code: the default, machine hierarchy (t_m), the hierarchy created on line 2 and split at lines 4 and 6 (t_2), and the hierarchy created at line 7 and split at line

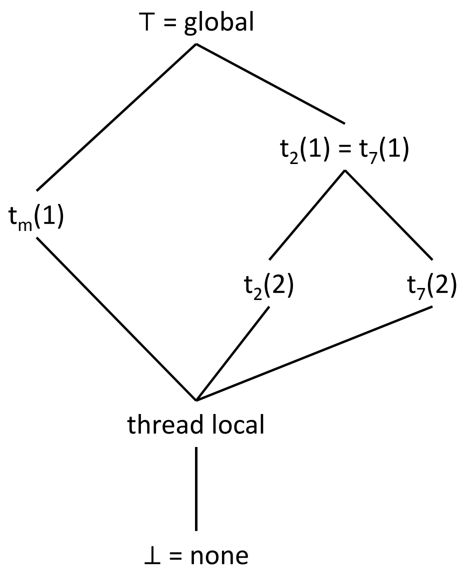


Figure 5.4: Example of a team lattice.

8 (t_7). Then `a` references the global team, the top level in the machine hierarchy $t_m(0)$. The team referenced by `b` is $t_2(0)$, the top level in team hierarchy t_2, which is equal to $t_m(0)$. Finally, `c` references the team $t_7(1)$, which is equal to $t_2(1)$. Since `c` is split in line 8, the team hierarchy t_7 has an additional level $t_7(2)$. Finally, all expressions in lines 1-5 are in the context of team $t_m(0)$, and those in lines 6-8 are in the context of team $t_2(1)$.

The root of each of the three team hierarchies is the global team, and the leaves are individual threads. Including these two levels, the machine team has three levels, with the second level (denoted by $t_m(1)$ using zero-indexing) **`local`**, meaning that all threads in a given team at that level share memory. Team t_2 has four levels, with the third level ($t_2(2)$) **`local`**, since it was created using a call to **`splitTeamSharedMem`**(). Team t_7 also has four levels and is equal to t_2 at the second level (i.e. $t_2(1) = t_7(1)$). The resulting team lattice is illustrated in Figure 5.4.

Chapter 6

Hierarchical Pointer Analysis

As discussed in §5.2, information about the pointers in a program is useful for many analyses and optimizations. In this chapter, we define a hierarchical pointer analysis that determines which allocation sites produce data that can be referenced by each variable in the program. The analysis also infers the possible threads on which the data may be located, using the team structure of the program to compute and report results. The pointer analysis we describe here is based on Andersen's analysis [6] and is an extension of a previous analysis that analyzes only the machine hierarchy [54, 49].

We begin by providing background on the analysis and defining a simple language, *Ti*, that abstracts away the irrelevant details of the Titanium language[1]. We then describe the abstract interpretation rules for a pointer analysis over *Ti* and prove correctness for the most complicated rule. Finally, we discuss how the analysis can be modified for locality inference as well as implementation of the analysis in Titanium.

6.1 Language

We start by defining a modified version of *Ti* [54], a basic SPMD language based on Titanium. Figure 6.1 illustrates the syntax of *Ti*. References have a width corresponding to the potential location of referenced data in the machine hierarchy. This width can range from 1 to the depth of the hierarchy d. Since *Ti* supports an arbitrary machine hierarchy, it follows the hierarchical partitioned global address space (HPGAS) memory model.

In addition to the machine hierarchy, we assume that there is a statically known set of team hierarchies arranged in a lattice, as described in §5.5.1. Elements in this lattice are named by a team hierarchy t and a level n.

The complete *Ti* language is as follows. Types can be integers or reference types. As mentioned above, the latter are parameterized by a width n in the range $[1, d]$. Expressions in *Ti* consist of the following:

[1]Though *Ti* follows the recursive single-program, multiple data (RSPMD) model of parallelism, the analysis can be extended to other models of parallelism. We do not do so, however, in this work.

- integer literals (n)
- variables (x). We assume a fixed set of variables of predefined type. We also assume that variables are private to each machine[2].
- reference allocations ($\texttt{new}_l\ \tau$). The expression $\texttt{new}_l\ \tau$ allocates a memory cell of type τ and returns a reference to the cell. Each allocation site has a unique label l.
- dereferencing ($*e$)
- type conversions ($\texttt{convert}(e, t, n)$). This asserts that the given expression evaluates to a reference located on a machine that is in the same subteam as the executing machine in the team hierarchy and level specified by t and n.
- communication ($\texttt{transmit}\ e_1\ \texttt{from}\ e_2\ \texttt{in}\ (t, n)$). The expression $\texttt{transmit}\ e_1\ \texttt{from}\ e_2\ \texttt{in}\ (t, n)$ evaluates e_1 on machine e_2 in the subteam specified by the hierarchy t and level n and transmits the result to all other machines in the subteam[3].
- sequencing ($e_1; e_2$)
- assignment to variables ($x := e$)
- assignment through references ($e_1 \leftarrow e_2$). In $e_1 \leftarrow e_2$, e_2 is written into the location referred to by e_1.

For simplicity, *Ti* does not have conditional statements. Since the analysis is flow-insensitive, conditionals are not essential to it.

Only the `transmit` and `convert` expressions differ from the previous version of *Ti*. The `transmit` expression now takes in a statically known team hierarchy as well as an integer denoting the level in the given team hierarchy over which the expression executes, with 0 denoting the top level. As such, this new version of *Ti* has an RSPMD model of parallelism. The `convert` expression also now takes in a team hierarchy and level. This level argument matches that of `transmit` in that 0 is the top level, unlike in the `convert` expression of the original *Ti* language.

6.1.1 Type System

The type checking rules for *Ti* are summarized in Figure 6.3. The rules for integer literals, variables, sequencing, and variable assignments are straightforward.

As in the approach of Liblit and Aiken, [65], we define an $expand$ function and a $robust$ predicate to facilitate type checking. The $expand$ function widens a type when necessary, and the $robust$ predicate determines when it is legal to assign to a reference. These functions are shown

[2]Throughout this chapter, we will use *machine* interchangeably with *thread*.

[3]Though a `transmit` operation is equivalent to a broadcast in Titanium, we use `transmit` instead of `broadcast` as it is a modification of the `transmit` expression described by Liblit and Aiken [65].

$$\begin{aligned}
n &::= \text{integer literals} \\
x &::= \text{variables} \\
t &::= \text{team hierarchies} \\
\tau &::= \mathit{int} \mid \texttt{ref}_n\ \tau && \text{(types)} \\
e &::= n \mid x \mid \texttt{new}_l\ \tau \mid\ *e \mid \texttt{convert}(e, t, n) \\
&\quad \mid \texttt{transmit}\ e_1\ \texttt{from}\ e_2\ \texttt{in}\ (t, n) \\
&\quad \mid e_1; e_2 \mid x := e \mid e_1 \leftarrow e_2 && \text{(expressions)}
\end{aligned}$$

Figure 6.1: The syntax of the *Ti* language.

$$\mathit{expand}(\tau, n) \equiv \begin{cases} \texttt{ref}_{\mathit{max}(n', n)}\ \tau' & \text{if } \tau = \texttt{ref}_{n'}\tau' \\ \tau & \text{otherwise} \end{cases}$$

$$\mathit{robust}(\tau, n) \equiv \begin{cases} \mathit{false} & \text{if } \tau = \texttt{ref}_{n'}\tau' \ \wedge\ n' < n \\ \mathit{true} & \text{otherwise} \end{cases}$$

Figure 6.2: Type manipulating functions.

in Figure 6.2. Two other functions are useful. The function $lat(t, n)$ converts a team hierarchy and level into the corresponding element in the team lattice. The function $locality(h)$ returns the locality of a given team-lattice element, resulting in a value in the range $[1, d]$ corresponding to a width in the machine hierarchy.

The allocation expression $\texttt{new}_l\ \tau$ produces a reference type $\texttt{ref}_1\ \tau$ of width 1, since the allocated memory is guaranteed to be on the machine that is performing the allocation. Pointer dereferencing is more problematic, however. Consider the situation in Figure 6.4, where x on machine 0 refers to a location on machine 0 that refers to a location on machine 1. This implies that x has type $\texttt{ref}_1\ \texttt{ref}_2\ \tau$. The result of $*x$ should be a reference to the location on machine 1, so it must have type $\texttt{ref}_2\ \tau$. In general, a dereference of a value of type $\texttt{ref}_a\ \texttt{ref}_b\ \tau$ produces a value of type $\texttt{ref}_{max(a,b)}\ \tau$.

The `convert` expression asserts that the given expression evaluates to a location on a machine in the same subteam in the given team hierarchy and level as the executing machine. Thus implies that the distance between the two machines is at most one less than the locality of the specified subteam. (As described in §5, distance is zero-indexed but locality is one-indexed.) A programmer can use `convert` expressions to inform the compiler that the reference is to data residing on a machine closer than the original width, such as after a dynamic check that this is the case. The resulting type is the same as the input expression, but with a new width matching the locality of

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{new}_l\, \tau : \texttt{ref}_1\, \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{ref}_n\, \tau}{\Gamma \vdash *e : \mathit{expand}(\tau, n)}$$

$$\frac{\Gamma \vdash e : \texttt{ref}_n\, \tau}{\Gamma \vdash \texttt{convert}(e, t, n') : \texttt{ref}_{\mathit{locality}(\mathit{lat}(t,n'))}\, \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{transmit}\, e_1\, \texttt{from}\, e_2\, \texttt{in}\, (t, n) : \mathit{expand}(\tau, \mathit{locality}(\mathit{lat}(t, n)))}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{ref}_n\, \tau \quad \Gamma \vdash e_2 : \tau \quad \mathit{robust}(\tau, n)}{\Gamma \vdash e_1 \leftarrow e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{ref}_n\, \tau \quad n < n'}{\Gamma \vdash e : \texttt{ref}_{n'}\, \tau}$$

Figure 6.3: Type checking rules.

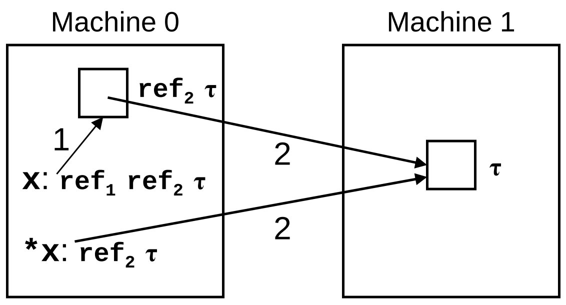


Figure 6.4: Dereferences may require width expansion. The arrow labels correspond to pointer widths.

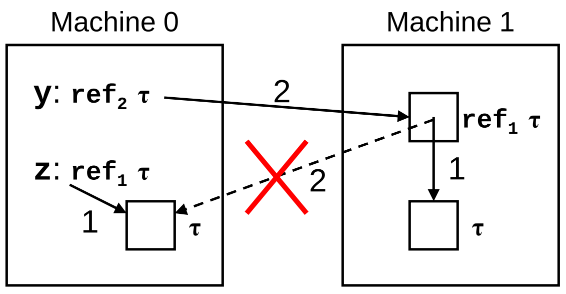


Figure 6.5: The assignment $y \leftarrow z$ is forbidden, since the location referred to by y can only hold pointers of width 1 but requires a pointer of width 2 to refer to z.

the given subteam.

In the `transmit` expression, if the value to be communicated is an integer, then the resulting type is still an integer. If the value is a reference, however, the width of the result must be set to the maximum of its original width and the locality of the subteam over which the `transmit` operation executes.

The typing rule for the assignment through reference expression is also nontrivial. Consider the case where y has type $\texttt{ref}_2\ \texttt{ref}_1\ \tau$, as in Figure 6.5. Should it be possible to assign to y with a value of type $\texttt{ref}_1\ \tau$? Such a value must be on machine 0, but the location referred to by x is on machine 1. Since that location holds a value of type $\texttt{ref}_1\ \tau$, it must refer to a location on machine 1. Thus, the assignment should be forbidden. In general, an assignment to a reference of type $\texttt{ref}_a\ \texttt{ref}_b\ \tau$ should only be allowed if $a \leq b$.

There is also a subtyping rule that allows for implicit widening of a reference. Subsumption is only allowed for the top-level width of a reference.

6.1.2 Operational Semantics

We now describe the operational semantics of *Ti*. First, we define the following semantic domains and naming conventions for their elements:

M	(the set of machines)	$m \in M$	(a machine)
H	(the set of team-lattice elements)	$h \in H$	(a team-lattice element)
A	(the set of local addresses)	$a \in A$	(a local address)
Id	(the set of identifiers)		
N	(the set of integer literals)	$n \in N$	(an integer)
$Var = M \times Id$	(the set of variables)		
L	(the set of allocation-site labels)	$l \in L$	(a label)
T	(the set of all team hierarchies)	$t \in T$	(a team hierarchy)
U	(the set of all types)	$\tau \in U$	(a type)
$G = L \times M \times A$	(the set of global addresses)	$g = (l, m, a) \in G$	(a global address)
$V = N \cup G$	(the set of values)	$v \in V$	(a value)
$Store = (G \cup Var) \to V$	(the contents of memory)	$\sigma \in Store$	(a memory state)
Exp	(the set of all expressions)	$e \in Exp$	(an expression)

Judgments in our operational semantics have the form $\langle e, m, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, which means that expression e executed on machine m in a global state σ evaluates to the value v and results in the new state σ'. We use the notation $\sigma[g := v]$ to denote the function $\lambda x.\ \texttt{if}\ x = g\ \texttt{then}\ v\ \texttt{else}\ \sigma(x)$.

Only the rules for `convert` and `transmit` change from the previous version of *Ti*. However, we provide all the rules for completeness. The rules for integer and variable expressions are trivial.

$$\frac{}{\langle n, m, \sigma\rangle \Downarrow \langle n, \sigma\rangle} \qquad \frac{}{\langle x, m, \sigma\rangle \Downarrow \langle \sigma(x), \sigma\rangle}$$

For allocations, we introduce a special $null$ value to represent uninitialized pointers. The result of an allocation is an address on the local machine that is guaranteed to not already be in use.

$$\frac{}{\langle new_l\ \tau, m, \sigma\rangle \Downarrow \langle (l, m, a), \sigma[(l, m, a) := null]\rangle}\ (a \text{ is fresh on } m)$$

The rule for dereferencing is simple, except that it is illegal to dereference a $null$ pointer.

$$\frac{\langle e, m, \sigma\rangle \Downarrow \langle g, \sigma'\rangle \quad g \neq null}{\langle *e, m, \sigma\rangle \Downarrow \langle \sigma'(g), \sigma'\rangle}$$

The rule for variable assignment is also simple.

$$\frac{\langle e, m, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle x := e, m, \sigma\rangle \Downarrow \langle v, \sigma'[x := v]\rangle}$$

The rule for assignment through a reference is the combination of a dereference and a normal assignment.

$$\frac{\langle e_1, m, \sigma\rangle \Downarrow \langle g, \sigma_1\rangle \quad \langle e_2, m, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle \quad g \neq null}{\langle e_1 \leftarrow e_2, m, \sigma\rangle \Downarrow \langle v, \sigma_2[g := v]\rangle}$$

The rule for sequencing is as expected.

$$\frac{\langle e_1, m, \sigma\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \langle e_2, m, \sigma_1\rangle \Downarrow \langle v_2, \sigma_2\rangle}{\langle e_1; e_2, m, \sigma\rangle \Downarrow \langle v_2, \sigma_2\rangle}$$

The `convert` expression is now generalized to assert that the given expression references a location on a machine that is in the same subteam as the executing machine at the given level of the given team hierarchy. (Providing the machine hierarchy and inverting the level argument results in the old `convert` expression.) We make use of the $team_c$ function, which takes in a machine m, a team hierarchy t, and an integer level n and returns the set of machines that are in the same subteam as m at level n in t.

$$\frac{\langle e, m, \sigma\rangle \Downarrow \langle g = (l, m', a), \sigma'\rangle \quad m' \in team_c(m, t, n)}{\langle \texttt{convert}(e, t, n), m, \sigma\rangle \Downarrow \langle g, \sigma'\rangle}$$

For the `transmit` operation, the expression is evaluated on the given machine in the context of the given team. The function $size$ takes in a machine m, a team hierarchy t, and an integer level n. It returns the size of m's subteam in t at level n. The function $mach$ takes in a team-relative machine ID i in addition to those arguments. It returns the machine assigned the ID i in the specified subteam.

$$\frac{\langle e_2, m, \sigma\rangle \Downarrow \langle i, \sigma_2\rangle \quad i \in [0, size(m, t, n)) \quad \langle e_1, mach(m, t, n, i), \sigma_2\rangle \Downarrow \langle v, \sigma_1\rangle}{\langle \texttt{transmit}\ e_1\ \texttt{from}\ e_2\ \texttt{in}\ (t, n), m, \sigma\rangle \Downarrow \langle v, \sigma_1\rangle}$$

We now turn our attention to a pointer analysis for the new version of the *Ti* language. So that we can ignore any issues of concurrency and also for efficiency, our analysis is flow-insensitive. We only define the analysis on the single machine m – since *Ti* is SPMD, the results are the same for all machines.

6.2 Concrete Domain

Since our analysis is flow-insensitive, we need not determine the concrete state at each point in a program. Instead, we define the concrete state over the whole program. Since we are doing pointer analysis, we are only interested in reference values, and since a location can contain different values over the lifetime of the program, we must compute the set of all possible values for each memory location and variable on machine m. The concrete state thus maps each memory location and variable to a set of memory locations, and it is a member of the domain $CS = (G + Id) \to \mathcal{P}(G)$.

6.3 Abstract Domain

For our abstract semantics, we define an *abstract location* to correspond to the abstraction of a concrete memory location. Abstract locations are defined relative to a particular machine m. An abstract location relative to machine m is a member of the domain $A_m = L \times H$ – it is identified by both an allocation site and a team-lattice element. An element a_1 of A_m is subsumed by another element a_2 if a_1 and a_2 have the same allocation site, and the team of a_2 subsumes the team of a_1. The elements of A_m are thus ordered by the following relation:

$$(l, h_1) \sqsubseteq (l, h_2) \iff h_1 \sqsubseteq h_2$$

The ordering thus has height d_{lat}, the height of the team lattice.

We define a *points-to set* S as a mapping between allocation sites and team-lattice elements, $S : L \to H$. We use R to denote the set of all points-to sets, with $|R| = |H|^{|L|}$, and define the following ordering on R, for all $S_1, S_2 \in R$:

$$S_1 \sqsubseteq S_2 \iff \forall l \in L.\ S_1(l) \sqsubseteq S_2(l)$$

The points-to set S_1 is subsumed by S_2 if for every allocation site, its team element in S_1 is subsumed by its team element in S_2. This ordering results in a lattice with the following minimal and maximal elements:

$$\forall l \in L.\ S_\bot(l) = \bot$$
$$\forall l \in L.\ S_\top(l) = \top$$

The maximal chain between $S_\bot$ and $S_\top$ is derived by increasing the team element for one allocation site at a time along the maximal chain in H, so the lattice has height $d_{lat} \cdot |L| + 1$.

We define a $team_a$ function, similar to $team_c$, that takes in a machine m and team lattice element h and returns the set of machines in the same team as m in the team hierarchy and level denoted by h. In particular, $team_a(m, \top) = M$, $team_a(m, \bot) = \emptyset$, and $team_a(m, \text{thread local}) = \{m\}$ for any machine m.

We now define a Galois connection between $\mathcal{P}(G)$ and R as follows:

$$\begin{aligned}\gamma_m(S) &= \big\{(l, m', a) \mid m' \in team_a(m, S(l))\big\}\\ \alpha_m(C) &= \sqcap\big\{S \mid C \sqsubseteq \gamma_m(S)\big\}\end{aligned}$$

The concretization of an abstract location (l, n) with respect to machine m is the set of all concrete locations with the same allocation site and located on machines that are in the same team as m in the team hierarchy and level specified by $S(l)$. The abstraction with respect to m of a concrete location (l, m', a) is an abstract location with the same allocation site and the team-lattice element that is the join of all elements for which m and m' are in the same subteam.

Finally, we abstract the concrete domain CS to the following abstract domain, which maps abstract locations and variables to points-to sets of abstract locations:

$$AS = (A_m + Id) \to R$$

An element σ_A of AS is subsumed by σ'_A if the points-to set of each abstract location and variable in σ_A is subsumed by the corresponding set in σ'_A. The elements of AS are therefore ordered as follows:

$$\sigma_A \sqsubseteq \sigma'_A \iff \forall x \in (A_m + Id).\ \sigma_A(x) \sqsubseteq \sigma'_A(x)$$

The resulting lattice has height in $\mathrm{O}(d_{lat} \cdot |L| \cdot (|A_m| + |Id|)) = \mathrm{O}(d_{lat} \cdot |L| \cdot (d_{lat} \cdot |L| + |Id|))$. Since the number of allocation sites and identifiers is limited by the size of the input program P, the height is in $\mathrm{O}(d_{lat}^2 \cdot |P|^2)$.

6.4 Abstract Semantics

For each expression in *Ti*, we provide inference rules for how the expression updates the abstract state σ_A. The judgments are of the form $\langle e, \sigma_A\rangle \Downarrow \langle S, \sigma'_A\rangle$, which means that expression e in abstract state σ_A can refer to the abstract locations in the points-to set S and results in the modified abstract state σ'_A. We use the notation $\sigma[g := v]$ to denote the function $\lambda x.\ \texttt{if}\ x = g\ \texttt{then}\ v\ \texttt{else}\ \sigma(x)$. Most of the rules are derived directly from the operational semantics of the language.

The rules for integer, team, and variable expressions are straightforward. None of them updates the abstract state, and the latter returns the points-to set of the variable.

$$\frac{}{\langle n, \sigma_A\rangle \Downarrow \langle S_\bot, \sigma_A\rangle} \qquad \frac{}{\langle t, \sigma_A\rangle \Downarrow \langle S_\bot, \sigma_A\rangle} \qquad \frac{}{\langle x, \sigma_A\rangle \Downarrow \langle \sigma_A(x), \sigma_A\rangle}$$

An allocation results in a thread local reference. It makes use of the $only$ function, which takes in an allocation site l and a team-lattice element h and returns a points-to set Q such that

$$\forall k \in L.\ (k = l \implies Q(k) = h) \wedge (k \neq l \implies Q(k) = \bot).$$

$$\frac{}{\langle new_l\ \tau, \sigma_A\rangle \Downarrow \langle only(l, \text{thread local}), \sigma_A\rangle}$$

The rule for dereferencing is similar to the operational semantics rule, except that all source abstract locations are simultaneously dereferenced.

$$\frac{\langle e, \sigma_A\rangle \Downarrow \langle S, \sigma'_A\rangle}{\langle *e, \sigma_A\rangle \Downarrow \langle \bigsqcup_{l \in L} \sigma'_A(S(l)), \sigma'_A\rangle}$$

The rule for sequencing is also analogous to its operational semantics rule.

$$\frac{\langle e_1, \sigma_A\rangle \Downarrow \langle S_1, \sigma'_A\rangle \quad \langle e_2, \sigma'_A\rangle \Downarrow \langle S_2, \sigma''_A\rangle}{\langle e_1; e_2, \sigma_A\rangle \Downarrow \langle S_2, \sigma''_A\rangle}$$

The rule for variable assignment merely updates the points-to set of the target variable with the source abstract locations.

$$\frac{\langle e, \sigma_A\rangle \Downarrow \langle S, \sigma'_A\rangle}{\langle x := e, \sigma_A\rangle \Downarrow \langle S, \sigma'_A[x := \sigma'_A(x) \sqcup S]\rangle}$$

The `convert` expression can only succeed if the result is located on a machine within the specified subteam, so it narrows all abstract locations to be within that subteam. It uses the $narrow$ function, which takes in a points-to set S, a team hierarchy t, and a level n and returns a points-to set Q such that

$$\forall l \in L.\ Q(l) = \begin{cases} lat(t, n) & \text{if } S(l) \not\sqsubseteq lat(t, n) \\ S(l) & \text{if } S(l) \sqsubseteq lat(t, n), \end{cases}$$

where $lat(t, n)$ is the team-lattice element corresponding to level n in t.

$$\frac{\langle e, \sigma_A\rangle \Downarrow \langle S, \sigma'_A\rangle}{\langle \texttt{convert}(e, t, n), \sigma_A\rangle \Downarrow \langle narrow(S, t, n), \sigma'_A\rangle}$$

The SPMD model of parallelism in *Ti* implies that the source expression of the `transmit` operation evaluates to abstract locations with the same labels on both the source and destination machines. The operation occurs within the context of the given team t and level n, so the source of the `transmit` is a machine m' in the same subteam as the executing machine m. Now let S be the source points-to set, which is the same on all machines since *Ti* is SPMD. Suppose $S(l) = lat(t', n')$ for some allocation site l. Then on machine m', the referenced location must be on a machine m'' in the same subteam as m' at level n' in team t'. Machine m is *not* necessarily in that same subteam, but it is in the same subteam as m'' in the the least common ancestor of $t(n)$ and $t'(n')$. Thus, from the point of view of m, the source abstract location must be widened to the team-lattice element corresponding to this least upper bound. The $widen$ function does so, taking in a points-to set S, a team hierarchy t, and a level n and returning a points-to set Q such that

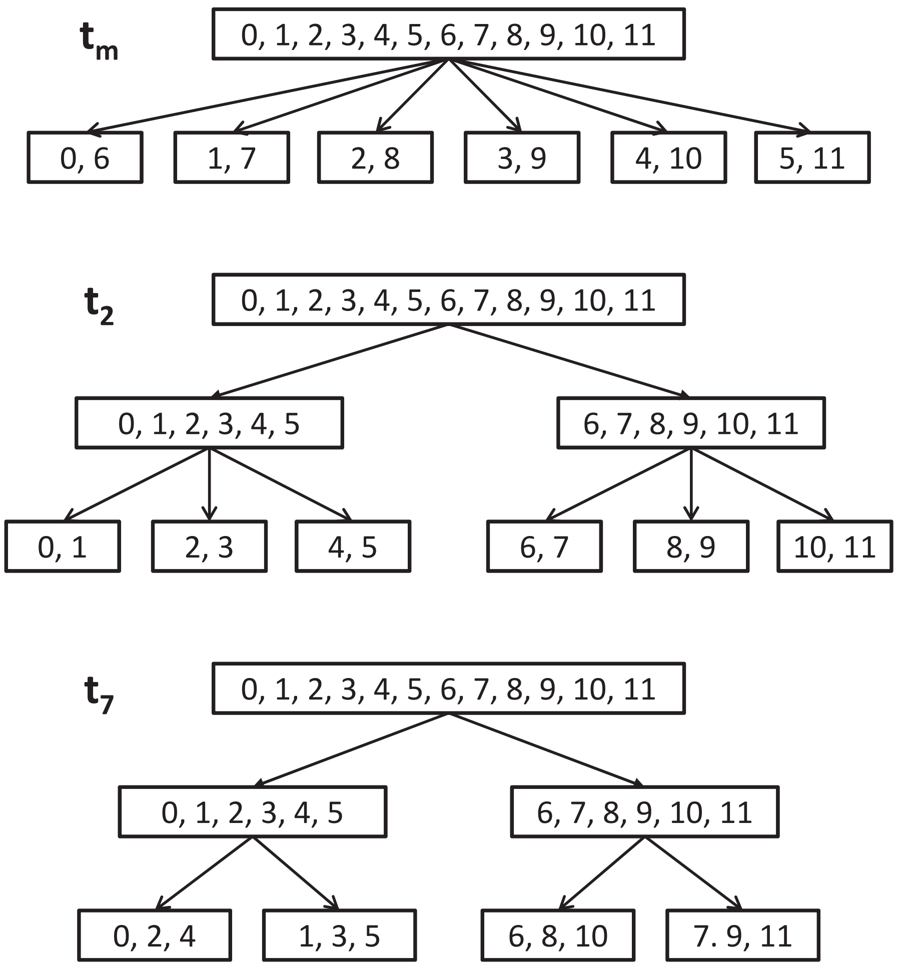


Figure 6.6: Sample team hierarchies.

$$\forall l \in L.\ Q(l) = \begin{cases} S(l) \sqcup lat(t,n) & \text{if } S(l) \neq \bot \\ \bot & \text{if } S(l) = \bot. \end{cases}$$

The rule for `transmit` then is

$$\frac{\langle e_2, \sigma_A\rangle \Downarrow \langle S_2, \sigma'_A\rangle \quad \langle e_1, \sigma'_A\rangle \Downarrow \langle S_1, \sigma''_A\rangle}{\langle \texttt{transmit}\ e_1\ \texttt{from}\ e_2\ \texttt{in}\ (t,n), \sigma_A\rangle \Downarrow \langle widen(S_1,t,n), \sigma''_A\rangle}$$

The rule for assignment through reference is the most interesting. Consider the team lattice in Figure 5.4. Suppose that the corresponding teams are as in Figure 6.6. (Though this set of teams would require different code to construct than that in §5.5.1, the resulting team lattice is the same. However, locality differs for some of the teams.) Now suppose an abstract location $a_2 = (l_2, t_7(2))$ is assigned into an abstract location $a_1 = (l_1, t_2(2))$ on machine 0. The abstract location a_1 can correspond to a concrete location on machine 1, since machines 0 and 1 are in the same subteam in $t_2(2)$, and a_2 can correspond to machine 2, since machines 0 and 2 are in the same subteam in $t_7(2)$. This implies that a concrete location corresponding to $(l_1, t_2(2))$, relative to machine 0, can point to a concrete location corresponding to $(l_2, t_7(t))$ relative to 0.

Now the same assignment occurs on machine 1, since *Ti* is SPMD. Relative to machine 1, a_1 can correspond to a concrete location on machine 1 and a_2 can correspond to a concrete location on machine 5. This implies that a concrete location corresponding to $(l_1, t_2(2))$, relative to *machine 0*, can point to a concrete location corresponding to $(l_2, t_2(1))$, relative to 0, since machines 0 and 5 aren't in the same subteam below $t_2(1)$. Thus, in order to account for this assignment on machine 1, it is $(l_2, t_2(1))$ that must be added to the points-to set of $(l_1, t_2(2))$.

The same assignment occurs on the rest of the machines, and we must update the points-to sets of (l_1, h) for all h in order to reflect the possible changes to the concrete state. As we will show below, whenever an assignment occurs from (l_2, h_2) to (l_1, h_1), we must update the points-to set of each (l_1, h_3) to include $(l_2, h_1 \sqcup h_2 \sqcup h_3)$. The update rule for assignment through reference is then

$$\frac{\langle e_1, \sigma_A\rangle \Downarrow \langle S_1, \sigma'_A\rangle \quad \langle e_2, \sigma'_A\rangle \Downarrow \langle S_2, \sigma''_A\rangle}{\langle e_1 \leftarrow e_2, \sigma_A\rangle \Downarrow \langle S_2, \mathit{update}(\sigma''_A, S_1, S_2)\rangle},$$

with update defined as

$$\begin{aligned}
&\mathit{update}(\sigma, S_1, S_2) = \\
&\quad \lambda(k) : A_m + Id\ . \\
&\qquad \texttt{if } (k \in Id \vee S_1(\mathit{label}(k)) = \bot) \texttt{ then } \sigma(k) \\
&\qquad \texttt{else } \sigma(k) \sqcup \left(\bigsqcup_{S_2(l_2) \neq \bot} \mathit{only}(l_2, S_1(\mathit{label}(k)) \sqcup S_2(l_2) \sqcup \mathit{team}(k)) \right)
\end{aligned}$$

and the label and team functions returning the corresponding components of an input abstract location.

6.5 Soundness

An abstract interpretation is sound if the abstraction and concretization functions are monotonic and form a Galois connection, and the abstract inference rules for each operation is correct. The former condition was shown in §6.3.

Most of the abstract inference rules are derived directly from the operational semantics, so their correctness is obvious. The rule for assignment through a reference, however, is nontrivial, so we prove its correctness here.

Let a_i^m represent the abstract location a_i with respect to machine m. Let h^m represent a team-lattice element h with respect to m.

Consider an assignment $e_1 \leftarrow e_2$. Let m be the reference machine for the analysis. Without loss of generality, assume that e_1 evaluates to the lone abstract location $a_1^m = (l_1, h_1^m)$, and that e_2 evaluates to $a_2^m = (l_2, h_2^m)$. Consider the execution of this assignment on the following machines:

- On machines m' such that m and m' are in the same subteam in the team hierarchy and level represented by h_1^m. This implies that abstract locations with team element h_1^m are the same with respect to both m and m', i.e. $a_1^{m'} = a_1^m$. Thus, the assignment can target any concrete location corresponding to a_1^m.

  Now suppose that $h_2^m \sqsubset h_1^m$. Then the $a_2^{m'}$ are not equivalent for all machines m'. However, the $a_2^{m'}$ for a particular m' contains the concrete locations (l_2, m', a) for any a. Considering the assignment on all machines m', the concrete locations in a_1^m can receive any of the source concrete locations (l_2, m', a) for all m' and a. This set of source locations corresponds exactly to the abstract location $a_{2'}^m = (l_2, h_1^m)$.

  Suppose instead that $h_2^m \sqsupseteq h_1^m$. Then the machines m' all agree on the set $a_2^{m'} = a_2^m$. Thus, regardless of which machine the assignment is executed on, the source locations correspond exactly to a_2^m.

  Finally, suppose that $h_2^m \not\sqsubset h_1^m$ and $h_2^m \not\sqsupseteq h_1^m$. Let $h_{2''}^m = h_1^m \sqcup h_2^m$. Then the machines m' all agree on the set $a_{2''}^{m'} = a_{2''}^m = (l_2, h_{2''}^m)$, since $h_1^m \sqsubseteq h_{2''}^m$. Furthermore, since $h_2^{m'} \sqsubseteq h_{2''}^{m'}$, it must be that $a_2^{m'} \sqsubseteq a_{2''}^{m'}$ for any machine m'. Since all machines m' agree on $a_{2''}^{m'}$, it contains all source locations on the machines m'. (It may also contain concrete locations that cannot be a source location on any machine m'. This does not affect soundness of the analysis, though it may result in loss of precision.)

  In any case, any of the concrete locations corresponding to a_1^m can now point to concrete locations corresponding to $a_{2*}^m = (l_2, h_1^m \sqcup h_2^m)$. To capture this in the abstract inference, it is sufficient to add a_{2*}^m to the points-to set of a_1^m. For consistency, a_{2*}^m should also be added to the points-to set of any abstract location $a_{1*}^m \sqsubseteq a_1^m$, since any of the concrete locations corresponding to a_{1*}^m can point to concrete locations corresponding to a_{2*}^m.

  Thus, it is sufficient to add the abstract location $a_{2*}^m = (l_2, h_1^m \sqcup h_2^m)$ to the points-to set of any $a_{1*}^m = (l_1, h_{1*}^m)$ such that $h_{1*}^m \sqsubseteq h_1^m$.

- On a machine m', where m and m' are not in the same subteam in the team hierarchy and level represented by h_1^m. The set of concrete locations corresponding to $a_1^{m'}$ all reside on machines in the same subteam as m' in that hierarchy and level. Let $h_{1'}^m$ correspond to the lowest level in that hierarchy in which m and m' are in the same subteam. Then $a_{1'}^{m'} = (l_1, h_{1'}^{m'}) = a_{1'}^m$, and since $h_1^{m'} \sqsubset h_{1'}^{m'}$, it is the case that $a_1^{m'} \sqsubset a_{1'}^{m'} = a_{1'}^m$, so that all concrete destination locations on machine m' are contained in $a_{1'}^m$.

  Now suppose $h_2^m \sqsubset h_{1'}^m$. Then all the concrete locations corresponding to $a_2^{m'}$ reside on machines in the same subteam as m in the hierarchy and level represented by $h_{1'}^m$, so that $a_2^{m'} \sqsubseteq a_{2'}^m$, where $a_{2'}^m = (l_2, h_{1'}^m)$. Thus, the source locations can be soundly approximated by $a_{2'}^m$.

  Suppose instead that $h_2^m \sqsupseteq h_{1'}^m$. Then m and m' agree on $a_2^{m'} = a_2^m$, so the source locations correspond to a_2^m.

Finally, suppose that $h_2^m \not\sqsubseteq h_{1'}^m$ and $h_2^m \not\sqsupseteq h_{1'}^m$. Let $h_{2''}^m = h_{1'}^m \sqcup h_2^m$. Then machines m and m' agree on the set $a_{2''}^{m'} = a_{2''}^m = (l_2, h_{2''}^m)$, since $h_{1'}^m \sqsubseteq h_{2''}^m$. Furthermore, since $h_2^{m'} \sqsubseteq h_{2''}^{m'}$, it is the case that $a_2^{m'} \sqsubseteq a_{2''}^{m'} = a_{2''}^m$, so that the latter contains all source locations on m'.

In any case, some of the concrete locations corresponding to $a_{1'}^m$ can now point to some of the concrete locations corresponding to $a_{2''}^m = (l_2, h_{1'}^m \sqcup h_2^m)$. Soundness can be maintained, though precision lost, if the analysis assumes that any concrete location corresponding to $a_{1'}^m$ can point to any concrete location corresponding to $a_{2''}^m$. Thus, $a_{2''}^m$ should be added to the points-to set of $a_{1'}^m$.

Considering all machines m' such that m and m' are not in the same subteam represented by h_1^m, none of the destination locations on any machine m' may reside on machines in the same subteam as m in h_1^m. Thus, no abstract locations $a_{1^*}^m \sqsubseteq a_1^m$ need to be updated. On the other hand, for any abstract location $a_{1^*}^m = (l_1, h_{1^*}^m) \sqsupset a_1^m$, there is at least one machine m' for which $h_{1^*}^m$ is the lowest level in its corresponding hierarchy for which m and m' are in the same subteam. Thus, the points-to set of each such abstract location $a_{1^*}^m$ must be updated to include $a_{2^*}^m = (l_2, h_{1^*}^m \sqcup h_2^m)$. Lastly, for any abstract location $a_{1^*}^m$ such that $a_{1^*}^m \not\sqsubseteq a_1^m$ and $a_{1^*}^m \not\sqsupset a_1^m$, all concrete locations in $a_{1^*}^m$ are contained in $a_{1^{**}}^m = (l_1, h_{1^*}^m \sqcup h_1^m)$. The latter points to $a_{2^{**}}^m = (l_2, h_{1^*}^m \sqcup h_1^m \sqcup h_2^m)$, so it is sound to update the points-to set of $a_{1^*}^m$ with $a_{2^{**}}^m$.

Combining the above, it is sufficient to add the abstract location $a_{2^{**}}^m = (l_2, h_{1^*}^m \sqcup h_1^m \sqcup h_2^m)$ to the abstract locations $a_{1^*}^m = (l_1, h_{1^*}^m)$ such that $h_{1^*}^m \not\sqsubseteq h_1^m$.

Summarizing over all possibilities, we obtain the rule that the abstract location $a_{2^{**}}^m = (l_2, h_{1^*}^m \sqcup h_1^m \sqcup h_2^m)$ is to be added to the points-to set of any $a_{1^*}^m = (l_1, h_{1^*}^m)$. This corresponds exactly to the update rule provided in §6.4.

6.6 Algorithm

The set of inference rules, instantiated over all the expressions in a program and applied in some arbitrary order[4], composes a function $F : AS \to AS$. Only the two assignment rules affect the input state σ_A, and in both rules, the output consists of a least-upper-bound operation involving the input state. As a result, F is a monotonically increasing function, and the least fixed point of F, $F_0 = \sqcup_n F^n(\lambda x.\ \emptyset)$, is the analysis result.

The function F has a rule for each program expression, so it takes time in $O(|P|)$ to apply it[5], where P is the input program. Since the lattice over AS has height in $O(d_{lat}^2 \cdot |P|^2)$, it takes time in $O(d_{lat}^2 \cdot |P|^3)$ to compute the fixed point of F.

[4]Since the analysis is flow-insensitive, the order of application is not important.

[5]We ignore the cost of the join operations here. In practice, representations of points-to sets tend to be small, so the cost of joining them can be neglected.

6.7 Locality Inference

One of the important applications of pointer analysis is inferring the locality of each of the variables in a program. A variable's *locality* corresponds to the lowest level in the machine hierarchy in which all concrete locations that the variable may point to reside on threads in the same subteam as the source thread. Since the original pointer analysis operated solely on the machine hierarchy, it was straightforward to determine the locality of a variable as the maximum level of any element in the variable's points-to set. In the new analysis, however, determining a variable's locality is more complicated.

Assuming the team lattice in Figure 5.4, consider a variable x with the points-to set $only(l, t_m(1))$. Suppose the assignment $x := e$ occurs where e has the points-to set $only(l, t_2(2))$. As discussed in §5.5.1, $t_m(1)$ and $t_2(2)$ are **`local`**, meaning that all threads within those subteams share physical memory. According to the abstract interpretation rule in §6.4, the new points-to set for x will be

$$only(l, t_m(1)) \sqcup only(l, t_2(2)) = only(l, global),$$

which is not **`local`**. The fact that the concrete state of x only includes pointers to **`local`** locations is no longer reflected in its abstract state.

To avoid this loss of information, we add a new entry $S(all)$ to every points-to set S. The value of $S(all)$ is always a machine hierarchy element corresponding to the lowest level in the machine hierarchy in which all concrete locations referenced by the points-to set must reside in the same subteam as the source thread. To facilitate computation of $S(all)$, every team hierarchy and level must have an associate locality, as discussed in §5.5.1. The new ordering on the set R of all points-to sets is

$$S_1 \sqsubseteq S_2 \Longleftrightarrow \forall l \in L.\ S_1(l) \sqsubseteq S_2(l) \wedge S_1(all) \sqsubseteq S_2(all).$$

We modify the abstract semantics in §6.4 to compute the value of $S(all)$ for each points-to set S. We make use of the $locality$ function that takes in a team-lattice element h and returns the locality of the corresponding subteam. Then the following modifications are made to the functions used in the abstract interpretation.

- $only(l, h)$ returns a points-to set S' such that
$$S'(all) = locality(h).$$
- $narrow(S, t, n)$ returns a points-to set S' such that
$$S'(all) = S(all) \sqcap locality(lat(t, n)).$$
- $widen(S, t, n)$ returns a points-to set S' such that
$$S'(all) = \begin{cases} S(all) \sqcup locality(lat(t, n)) & \text{if } S(all) \neq \bot \\ \bot & \text{if } S(all) = \bot. \end{cases}$$

The abstract interpretation rules otherwise remain the same, with join operations reflecting the new ordering on points-to sets.

6.8 Implementation

We have implemented a prototype of the pointer analysis in the Titanium compiler that analyzes only the machine team hierarchy. For evaluation purposes, we implemented three variants of the analysis, with one, two, and three levels of hierarchy. The single-level analysis combines all three levels and cannot be used for either locality or sharing inference. In two-level analysis, level 1 remains separate while levels 2 and 3 are combined. Level 1 must be separate in order to perform sharing analysis, and this separation still allows locality inference, though with less precision than combining levels 1 and 2. Finally, the three-level analysis separates all three levels, providing the most precise results.

6.8.1 Titanium Features

The *Ti* language is much simpler than Titanium, and certain Titanium features require special treatment:

- **types**: objects in Titanium have types, so the corresponding abstract locations are also typed.
- **fields**: objects can have multiple fields, so an abstract location must have points-to sets for each of its fields
- **arrays**: arrays can have multiple entries. For simplicity, the analysis makes no attempt to distinguish between the different entries of an array.
- **method calls**: methods may have parameters, return values, and a `this` value. The analysis considers each of these to be variables, and the result of a method call is the set of abstract locations corresponding to its return variable.
- **dynamic dispatch**: a method call on an object may dispatch to different targets at runtime. The analysis can compute a conservative but precise estimate of the possible dispatch targets by examining the types of the abstract locations corresponding to the source object.
- **native code**: native methods are handled conservatively for the most part. However, the analysis assumes that a native method does not violate type safety, and that it does not modify the fields of an object in certain ways. Native library methods are treated specially by the analysis if they violate these assumptions.

6.8.2 Optimizations

A handful of optimizations were applied to the pointer analysis. Execution time can likely be improved drastically by using binary decision diagrams [104].

6.8.2.1 Lazy Analysis

Titanium is to a large extent backwards compatible with Java, providing most of its language features and much of its library. The typical Titanium program uses only a small portion of the Java library, so analyzing the entire library is unnecessary. The pointer analysis implementation is lazy in that it only analyzes those methods that are reachable from the program entry point and static initializers. It does so by marking the `main()` method and static initializers reachable, and the rest of the methods unreachable. When the analysis encounters a call to an unreachable method, it makes the method reachable and proceeds to analyze it. This is continued until a fixed point is reached.

6.8.2.2 On-Demand Creation of Abstract Locations

Theoretically, the pointer analysis requires $A \cdot h$ abstract locations, where A is the number of allocation sites and h is the number of levels in the analysis. However, if a particular thread-local abstract location is never leaked beyond its creator thread, the analysis never uses the wider versions of the location. The implementation takes advantage of this fact by only creating the wider counterparts on demand if the thread-local version is leaked.

Chapter 7

Concurrency Analysis

A precise knowledge of the set of concurrent statements in parallel programs is fundamental to many analyses and optimizations, including race detection and synchronization elimination. In this chapter, we develop an analysis that determines which statements may be concurrent in the context of each team in a program. More specifically, for each level in a team hierarchy and each pair of statements, we infer if there is a subteam at that level that may contain two threads that execute the statements concurrently.

We start by reviewing a global concurrency analysis that we defined in previous work [52, 49]. This analysis constructs a graph representation of a program to represent its concurrency information. It is a flat analysis, in the sense that it only computes concurrency results for the global team, and it does not take into account the rest of the machine hierarchy or any of the other team hierarchies in a program. We then describe a hierarchical extension of the analysis that simultaneously computes concurrency information for all teams in a program.

7.1 Flat Concurrency Analysis

We begin by defining a flat, non-hierarchical concurrency analysis for SPMD programs with global, textually-aligned collectives. We start with a basic analysis and then improve on its results by only considering program paths that can occur at runtime.

7.1.1 Analysis Background

Our concurrency analyses do not operate directly on Titanium program's source code, but on a graph representation of a reduced form of the program, in order to simplify both the theory and implementation of the analyses.

7.1.1.1 Intermediate Language

We operate on an *intermediate language* that allows the full semantics of Titanium but is simpler to analyze. In particular, Titanium follows the object-oriented semantics of Java, including dy-

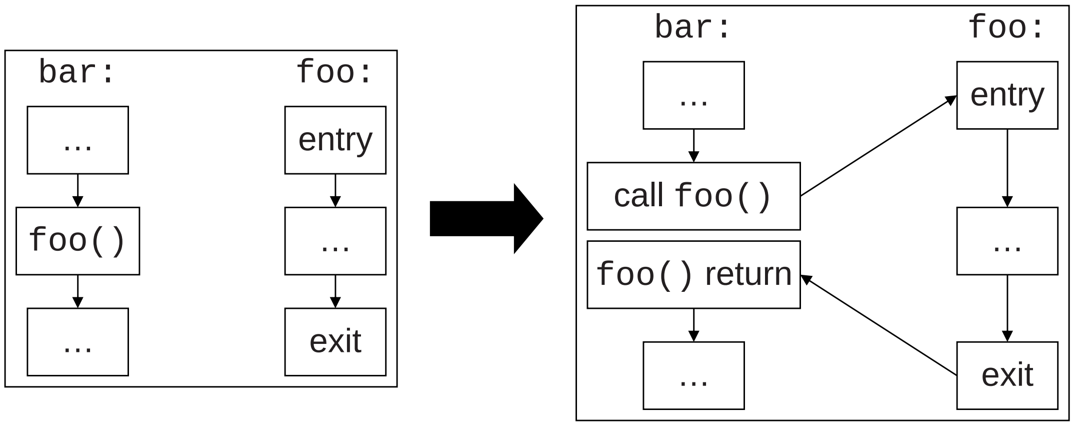


Figure 7.1: Construction of the interprocedural control-flow graph of a program from the individual method-flow graphs.

namic dispatches on instance methods. We rewrite such dynamic dispatches as conditionals for simplicity; a call `x.foo()`, where `x` is of type `A` in the class hierarchy

```
class A {
  void foo() { ... }
}

class B extends A {
  void foo() { ... }
}
```

gets rewritten to

```
if ([type of x is A])
  x.A$foo();
else if ([type of x is B])
  x.B$foo();
```

We also rewrite **`switch`** statements and conditional expressions (... ? ... : ...) as conditional **`if`** ... **`else`** ... statements.

7.1.1.2 Control-Flow Graphs

The concurrency algorithms are whole-program analyses that operate over a *control-flow graph* that represents the flow of execution in a program. Nodes in the graph correspond to expressions in the program, and a directed edge from one expression to another occurs when the target can execute immediately after the source.

The Titanium compiler produces an intraprocedural control-flow graph for each method in a program. We modify each of these graphs to model transfer of control between methods by splitting each method-invocation node into a call node and a return node. The incoming edges of the original node are attached to the call node, and the outgoing edges to the return node. An edge is added from the call node to the target method's entry node, and from the target method's exit node to the return node. (Since dynamic dispatches are rewritten as conditionals, each invocation only has a single target.) Figure 7.1 illustrates this procedure. We also add edges to model interprocedural control flow due to exceptions.

7.1.2 Basic Analysis

Titanium's structural correctness allows us to develop a simple graph-based algorithm for computing concurrent expressions in a program. The algorithm specifically takes advantage of Titanium's textually-aligned barriers and single-valued expressions. (We use the term *barrier* here to refer to not just actual barriers, but any collective operation that prevents code before and after the operation from running concurrently within the affected team. Thus, Titanium broadcast expressions do not qualify as barriers here, while exchange operations do. In this section, we assume all barriers are global.) For the purposes of the flat concurrency analysis, we assume that textual alignment is enforced either by the single type system described in §4.1 or the strict dynamic-alignment scheme discussed in §4.2.

The following definitions are useful in developing the analysis:

Definition 7.1.1. (Single Conditional) A *single conditional* is a conditional guarded by a single-valued expression.

As described in §4, a single-valued expression evaluates to coherent results on all threads. Thus, every thread is guaranteed to take the same branch of a single conditional. A single conditional may contain a barrier, since all threads are guaranteed to execute it, while a non-single conditional may not.

Definition 7.1.2. (Cross Edge) A *cross edge* in a control-flow graph connects the end of the first branch of a conditional to the start of the second branch.

Cross edges do not provide any control-flow information, since the second branch of a conditional does not execute immediately after the first branch. They are, however, useful for determining concurrency information, as shown in Theorem 7.1.4.

In order to determine the set of concurrent expressions in a program, we construct a *concurrency graph* G to represent concurrency information for the program P. We do so by inserting cross edges in the interprocedural control-flow graph of P for every non-single conditional and deleting all barriers and their adjacent edges. Algorithm 7.1.3 in Figure 7.2 illustrates this procedure. The algorithm runs in time $\mathrm{O}(n)$, where n is the number of statements and expressions in P, since it takes $\mathrm{O}(n)$ time to construct the control-flow graph of a program. The control-flow graph is very sparse, containing only $\mathrm{O}(n)$ edges, since the number of expressions that can execute immediately after a particular expression e is constant. Since at most n cross edges are added to the control-flow graph and at most $\mathrm{O}(n)$ barriers and adjacent edges are deleted, the resulting graph G is also of size in $\mathrm{O}(n)$.

The concurrency graph G allows us to determine the set of concurrent expressions using the following theorem:

```
Algorithm 7.1.3.
ConcurrencyGraph($P$ : program) : graph
  1. Let $G$ be the interprocedural control-flow graph of $P$, as described in §7.1.1.2.
  2. For each conditional $C$ in $P$ {
  3.    If $C$ is not a single conditional:
  4.       Add a cross edge for $C$ in $G$.
  5. } // End for (2).
  6. For each barrier $B$ in $P$:
  7.    Delete the node for $B$ and its adjacent edges from $G$.
  8. Return $G$.
```

Figure 7.2: Algorithm 7.1.3 computes the concurrency graph of a program by inserting cross edges into its control-flow graph and deleting all barriers.

```
1  Ti.barrier();
2  int i = 0;
3  int j = 1;
4  if (Ti.thisProc() < 5)
5    j += Ti.thisProc();
6  if (Ti.numProcs() >= 1) {
7    i = Ti.numProcs();
8    Ti.barrier();
9    j += i;
10 } else { j += 1; }
11 i = broadcast j from 0;
12 Ti.barrier();
13 j += i;
```

Code Phase	Statements
1	2, 3, 4, 5, 6, 7, 10, 11
8	9, 11
12	13

Figure 7.3: The set of code phases for a sample program.

Theorem 7.1.4. *Two expressions* a *and* b *in* P *can run concurrently only if one is reachable from the other in the concurrency graph* G.

In order to prove Theorem 7.1.4, we require the following definition:

Definition 7.1.5. (Code Phase) The *code phase* of a barrier is the set of expressions that may execute after the barrier but before hitting another barrier, including itself[1].

Figure 7.3 shows the code phases of an example program. Since each code phase is preceded by a barrier, and each thread must execute the same sequence of barriers, each thread executes the same sequence of code phases. This implies the following:

[1]A statement can be in multiple code phases, as is the case for a statement in a method called from multiple contexts.

Lemma 7.1.6. *Two expressions* a *and* b *in* P *can run concurrently only if they are in the same code phase.*

Proof. Suppose a and b are never in the same code phase. Then they are always preceded by two different barriers. Consider arbitrary occurrences of a and b in any program execution in which they both occur. (If one or both don't occur, then they trivially don't run concurrently.) Let B_a and B_b be the barriers preceding a and b, respectively. Since every thread executes the same set of barriers, either B_a precedes B_b or B_b precedes B_a. Since a occurs after B_a but before any other barrier, and b occurs after B_b but before any other barrier, this implies that a and b are separated by a barrier. Thus, a and b cannot run concurrently, since a barrier prevents the code before it and after it from executing concurrently. □

We can now prove Theorem 7.1.4:

Proof of Theorem 7.1.4. Suppose a and b can run concurrently. By Lemma 7.1.6, a and b must be in the same code phase S. By Definition 7.1.5, there must be program flows from the initial barrier B_S to a and b that do not go through barriers. There are three cases:

Case 1: There is a program flow from a to b in S. This means the control-flow graph of the program must contain a path from the node for a to the node for b that does not pass through a barrier. Since G contains all nodes and edges of the control-flow graph except those corresponding to barriers, it also contains such a path, so b is reachable from a.

Case 2: There is a program flow from b to a in S. This case is analogous to the one above.

Case 3: There is no program flow either from a to b or from b to a in S. Since there is a flow from B_S to a and from B_S to b, a and b must be in different branches of a conditional C. Since only one branch of a single conditional can run, C must be a non-single conditional in order for a and b to run concurrently. Without loss of generality, let a be in the first branch, and b be in the second. Since C is non-single, it cannot contain a barrier, and the end of the first branch is reachable in G from a without hitting a barrier. Similarly, b is reachable from the beginning of the second branch without executing a barrier. Since G contains a cross edge from the first branch of C to the second, this implies that there is a path from a to b in G that does not pass through a barrier. □

By Theorem 7.1.4, in order to determine the set of all pairs of concurrent expressions, it suffices to compute the pairs of expressions in which one is reachable from the other in the concurrency graph G. This can be done efficiently by performing a depth-first search from each expression in G. Algorithm 7.1.7 in Figure 7.4 does exactly this. The running time of the algorithm is dominated by the depth-first searches, each of which takes $\mathrm{O}(n)$ time, since G has at most n nodes and $\mathrm{O}(n)$ edges. At most n searches occur, so the algorithm runs in time $\mathrm{O}(n^2)$.

7.1.3 Feasible Paths

Algorithm 7.1.7 computes an over-approximation of the set of concurrent expressions. In particular, due to the nature of the interprocedural control-flow graph constructed in §7.1.1.2, the depth-first searches in Algorithm 7.1.7 can follow *infeasible paths*, paths that cannot structurally

```
Algorithm 7.1.7.
ConcurrentExpressions($P$ : program) : set
1. Let $concur \leftarrow \emptyset$.
2. Let $G \leftarrow$ ConcurrencyGraph($P$) [Algorithm 7.1.3].
3. For each access $a$ in $P$ {
4.    Do a depth-first search on $G$ starting from $a$.
5.    For each expression $b$ reached in the search:
6.       Insert $(a, b)$ into $concur$.
7. } // End for (3).
8. Return $concur$.
```

Figure 7.4: Algorithm 7.1.7 computes the set of all pairs of concurrent expressions in a given program.

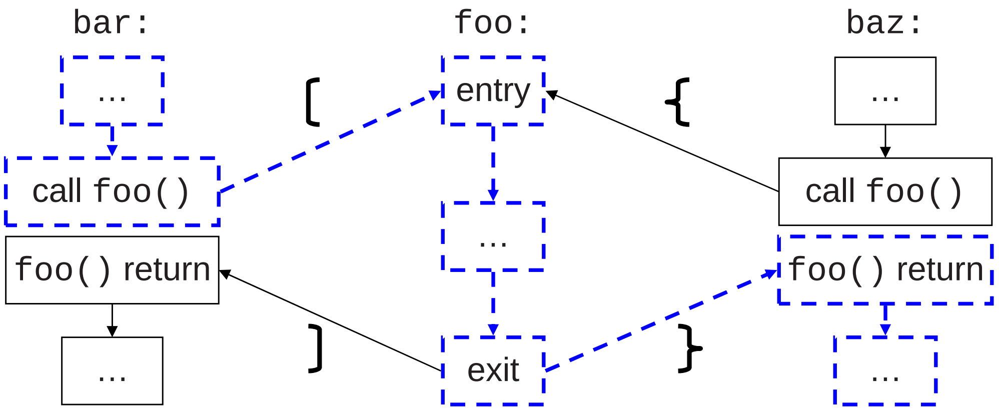


Figure 7.5: Interprocedural control-flow graph for two calls to the same function. The dashed path is infeasible, since `foo()` returns to a different context than the one from which it was called. The infeasible path corresponds to the unbalanced string "[}".

occur in practice. Figure 7.5 illustrates such a path, in which a method is entered from one context and exits into another.

In order to prevent infeasible paths, we follow the procedure outlined by Reps [86]. We label each method call edge and corresponding return edge with matching parentheses, as shown in Figure 7.5. Each path then corresponds to a string of parentheses composed of the labels of the edges in the path. A path is then infeasible, if in its corresponding string, an open parenthesis is closed by a non-matching parenthesis.

It is not necessary that a path's string be balanced in order for it to be feasible. In particular, two types of unbalanced strings correspond to feasible paths:

- A path with unclosed parentheses. Such a path corresponds to method calls that have not yet finished, as shown in the left side of Figure 7.6.

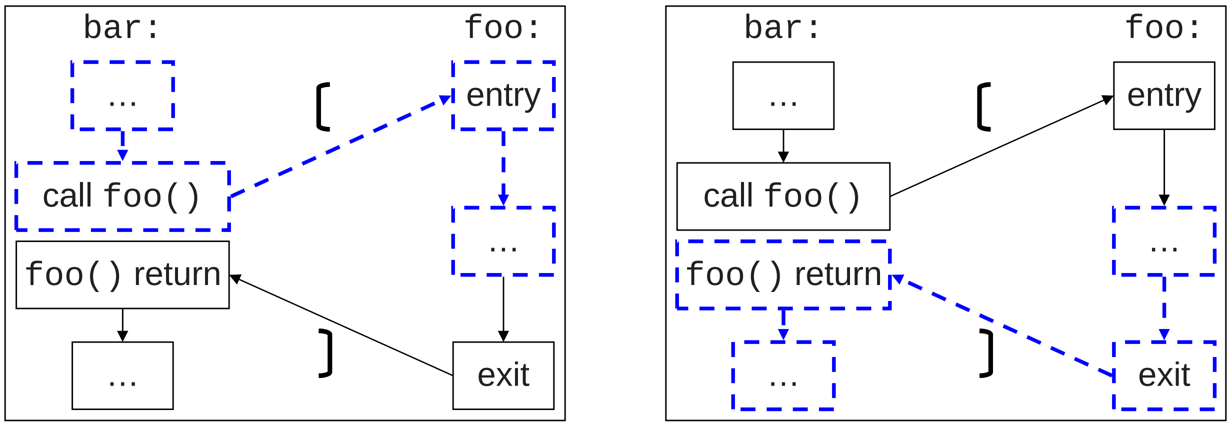


Figure 7.6: Feasible paths that correspond to unbalanced strings. The dashed path on the left corresponds to a method call that has not yet returned, and the one on the right corresponds to a path that starts in a method call that returns.

- A path with closing parentheses that follow a balanced prefix. Such a string is allowed since a path may start in the middle of a method call and corresponds to that method call returning, as shown in the right side of Figure 7.6.

Determining the set of nodes reachable[2] using a feasible path is the equivalent of performing context-free language (CFL) reachability on a graph using the grammar for each pair of matching parentheses $(_\alpha$ and $)_\alpha$. CFL reachability can be performed in cubic time for an arbitrary grammar [86]. Algorithm 7.1.7 takes only quadratic time, however, and we desire a feasibility algorithm that is also quadratic. In order to accomplish this, we develop a specialized algorithm that modifies the concurrency graph G and the standard depth-first search instead of using generic CFL reachability.

At first glance, it appears that a method must be revisited in every possible context in which it is called, since the context determines which open parentheses have been seen and therefore which paths can be followed. However, as we now show, the set of expressions that can be executed in a method call is the same regardless of context.

Theorem 7.1.8. *Ignoring the effect of the arguments, the set of expressions that may be executed in a call to a method* f *is the same regardless of the context in which* f *is called.*

Proof by induction over the function call depth.

Base case: The execution of f makes no method calls. Then the call to f can execute at most those expressions that are contained in f and reachable from its entry regardless of the calling context.

[2]In this section, we make no distinction between *reachable* and *reachable without hitting a barrier*. The latter reduces to the former if all barrier nodes are removed from each control-flow graph. We also restrict ourselves to a static definition of reachability, since reachability at runtime is uncomputable.

Inductive step: The execution of f makes method calls. By the inductive hypothesis[3], each method call in f can transitively execute the same expressions independent of the context. In addition, the call to f can execute exactly those expressions that are contained in f and reachable from its entry. The call to f thus can execute the same set of expressions regardless of context. □

Theorem 7.1.8 implies that the set of nodes reachable along a feasible path in a program's control-flow graph is independent of the context of a method call, with two exceptions:

- If a method call can complete, then the nodes after the call are reachable from a point before the call.
- If no context exists, such as in a search that starts from a point within a method f, then all nodes that can be reached from the return node of any method call to f are reachable.

The second case above can easily be handled by visiting a node twice: once in *some* context, and again in no context. The first case, however, requires adding bypass edges to the control-flow graph.

7.1.3.1 Bypass Edges

In §7.1.1.2, we constructed the interprocedural control-flow graph of a program by splitting each method call into a call node and a return node. We then added an edge from the call node to the target method's entry, and another from the target's exit to the return node. If the target's exit is reachable (or for our purposes, reachable without hitting a barrier) from the target's entry, then adding a *bypass edge* that connects the call node directly to the return node does not affect the transitive closure of the graph.

Computing whether or not a method's exit is reachable from its entry is not trivial, since it requires knowing whether or not the exits of each of the methods that it calls are reachable from their entries. Algorithm 7.1.9 in Figure 7.7 computes this by continually iterating over all the methods in a program, marking those that can complete through an execution path that only calls previously marked methods, until no more methods can be marked. In the first iteration of loop 3, it only marks those methods that can complete without making any calls, or equivalently, those methods that can complete using only a single stack frame. In the second iteration, it only marks those that can complete by calling only methods that don't need to make any calls, or equivalently, those methods that can complete using only two stack frames. In general, a method is marked in the ith iteration if it can complete using i, and no less than i, stack frames[4]. We now prove the correctness of Algorithm 7.1.9.

Theorem 7.1.10. *Algorithm 7.1.9 marks all methods that can complete using any number of stack frames.*

[3]In order for induction to be applicable, the function call depth in f must be finite. It is reasonable to assume that this is always the case, since in practice, an infinite function call depth is impossible due to finite memory limits.

[4]The fact that a method only requires a fixed number of stack frames does not mean that it can complete. A method may contain a loop with no exit, preventing the method from completing at all, or barriers along all paths through it, preventing it from completing without executing a barrier. Algorithm 7.1.9 does not mark such methods.

```
Algorithm 7.1.9.
ComputeBypasses($P$ : program, $G_1, \ldots, G_k$ : intraprocedural flow graph) : set
 1.  Let $change \leftarrow true$.
 2.  Let $marked \leftarrow \emptyset$.
 3.  While $change = true$ {
 4.    $change \leftarrow false$.
 5.    Set $visited(u) \leftarrow false$ for all nodes $u$ in $G_1, \ldots, G_k$.
 6.    For each method $f$ in $P$ {
 7.      If $f \notin marked$ and $CanReach(entry(f), exit(f), G_f, marked)$ {
 8.        $marked \leftarrow marked \cup \{f\}$.
 9.        $change \leftarrow true$.
10.      } // End if (7).
11.    } // End for (6).
12.  } // End while (3).
13.  Return $marked$.

14.  Procedure $CanReach(u, v$ : vertex, $G$ : graph, $marked$ : method set) : boolean:
15.    Set $visited(u) \leftarrow true$.
16.    If $u = v$:
17.      Return $true$.
18.    Else If $u$ is a method call to function $g$ and $g \notin marked$:
19.      Return $false$.
20.    For each edge $(u, w) \in G$ {
21.      If $visited(w) = false$ and $CanReach(w, v, G, marked)$:
22.        Return $true$.
23.    } // End for (20).
24.    Return $false$.
```

Figure 7.7: Algorithm 7.1.9 uses each method's intraprocedural control-flow graph (G_i) to determine if its exit is reachable from its entry.

Proof. Suppose there are some methods that can complete but that Algorithm 7.1.9 does not find. Out of these methods, let f be the one that can complete with the minimum number of stack frames j. In order for f to require j frames to complete, there must be an execution path through f that only calls methods that require at most $j - 1$ frames to complete. These methods must all be marked, since f is the minimum method that isn't marked. Let i be the iteration in which the last of these methods is marked. Since a method is marked in this iteration, loop 3 will iterate at least once more. Since f now has a path in which it only calls marked methods, f will be marked in the $(i + 1)$th iteration. This is a contradiction, so Algorithm 7.1.9 marks all methods that can complete. □

Algorithm 7.1.9 requires quadratic time to complete in the worst case. Each iteration of loop 3 visits at most n nodes. Only k iterations are necessary, where k is the number of methods in

```
Algorithm 7.1.11.
FeasibleSearch($v$ : vertex, $G$ : graph) : set
 1. Let $visited \leftarrow \emptyset$.
 2. Let $s \leftarrow \emptyset$.
 3. Call $FeasibleDFS(v, G, s, visited)$.
 4. Return $visited$.

 5. Procedure $FeasibleDFS(v$ : vertex, $G$ : graph, $s$ : stack, $visited$ : set):
 6.   If $s = \emptyset$ {
 7.     If $no\_context\_mark(v)$ return.
 8.     Set $no\_context\_mark(v) \leftarrow true$.
 9.   } // End if (6).
10.   Else {
11.     If $context\_mark(v)$ return.
12.     Set $context\_mark(v) \leftarrow true$.
13.   } // End else (10).
14.   $visited \leftarrow visited \cup \{v\}$
15.   For each edge $(v, u) \in G$ {
16.     Let $s' \leftarrow s$.
17.     If $label(v, u)$ is a close symbol and $s' \neq \emptyset$ {
18.       Let $o \leftarrow pop(s')$.
19.       If $label(v, u)$ does not match $o$:
20.         Skip to next iteration of 15.
21.     } // End if (17).
22.     Else if $label(v, u)$ is an open symbol:
23.       Push $label(v, u)$ onto $s'$.
24.     Call $FeasibleDFS(u, G, s')$.
25.   } // End for (15).
```

Figure 7.8: Algorithm 7.1.11 computes the set of nodes reachable from the start node through a feasible path.

the program, since at least one method is marked in all but the last iteration of the loop. The total running time is thus $\mathrm{O}(kn)$ in the worst case. In practice, only a small number of iterations are necessary[5], and the running time is closer to $\mathrm{O}(n)$.

After computing the set of methods that can complete, it is straightforward to add bypass edges to the concurrency graph G: for each method call c, if the target of c can complete, add an edge from c to its corresponding method return r. This can be done in time $\mathrm{O}(n)$.

7.1.3.2 Feasible Search

Once bypass edges have been added to the graph G, a modified depth-first search can be used to find feasible paths. A stack of open but not yet closed parenthesis symbols must be maintained, and an encountered closing symbol must match the top of this stack, if the stack is nonempty. In addition, as noted above, the modified search must visit each node twice, once in no context and once in *some* context. Algorithm 7.1.11 in Figure 7.8 formalizes this procedure, and we prove that it only follows feasible paths.

Theorem 7.1.12. *Algorithm 7.1.11 does not follow any infeasible paths.*

Proof. Consider an arbitrary infeasible path p. In order for p to be infeasible, the labels along p must form a string in which an open parenthesis $(_\alpha$ is closed by a non-matching parenthesis $)_\beta$. Consider the execution of Algorithm 7.1.11 on this path. An open parenthesis is pushed onto the the stack s when it is encountered, so before any close parentheses are encountered, the top of the stack is the most recently opened parenthesis. A close parenthesis causes the top of the stack to be popped, so in general, the top of the stack is the most recently opened parenthesis that has not yet been closed. Now consider s when the label $)_\beta$ is reached. The symbol $(_\alpha$ must be on the top of s, since $)_\beta$ closes it. But Algorithm 7.1.11 checks the top of the stack against the newly encountered label, and since they don't match, it does not proceed along p. □

Since G contains bypass edges and Algorithm 7.1.11 visits each node both in some context and in no context, it finds all nodes that can be reachable in a feasible path from the source. Since it visits each node at most twice, it runs in time $\mathrm{O}(n)$.

7.1.3.3 Feasible Concurrent Expressions

Putting it all together, we can now modify Algorithm 7.1.7 to find only concurrent expressions that are feasible. As in Algorithm 7.1.7, the concurrency graph G must first be constructed. Then the intraprocedural flow graphs of each method must be constructed, Algorithm 7.1.9 used to find the methods that can complete without hitting a barrier, and the bypass edges inserted into G. Then Algorithm 7.1.11 must be used to perform the searches instead of a vanilla depth-first search. Algorithm 7.1.13 in Figure 7.9 illustrates this procedure.

The setup of Algorithm 7.1.13 calls Algorithm 7.1.9, so it takes $\mathrm{O}(kn)$ time. The searches each take time in $\mathrm{O}(n)$, and at most n are done, so the total running time is in $\mathrm{O}(kn + n^2) = \mathrm{O}(n^2)$, quadratic as opposed to the cubic running time of generic CFL reachability.

7.2 Hierarchical Concurrency Analysis

We now develop a hierarchical extension to the above feasible-paths concurrency analysis. The preceding analysis determines concurrency information in the context of a single team, namely the

[5] Even on the largest example we tried (>45,000 lines of user and library code, 1226 methods), Algorithm 7.1.9 required only five iterations to converge.

```
Algorithm 7.1.13.
FeasibleConcurrentExpressions($P$ : program) : set
 1. Let $G \leftarrow$ ConcurrencyGraph($P$) [Algorithm 7.1.3].
 2. For each method $f$ in $P$ {
 3.   Construct the intraprocedural flow graph $G_f$ of $f$.
 4.   For each barrier $B$ in $f$ {
 5.     Delete $B$ from $G_f$.
 6.   } // End for (4).
 7. } // End for (2).
 8. Let $bypass \leftarrow$ ComputeBypasses($P, G_1, \ldots, G_k$) [Algorithm 7.1.9].
 9. For each method call and return pair $c, r$ in $P$ {
10.   If the target $f$ of $c, r$ is in $bypass$:
11.     Add an edge $(c, r)$ to $G$.
12. } // End for (9).
13. For each expression $a$ in $P$ {
14.   Let $visited \leftarrow$ FeasibleSearch($a$, $G$) [Algorithm 7.1.11].
15.   For each expression $b \in visited$:
16.     Insert $(a, b)$ into $concur$.
17. } // End for (13).
18. Return $concur$.
```

Figure 7.9: Algorithm 7.1.13 computes the set of all concurrent expressions that can feasibly occur in a given program.

global team. A trivial way to extend the analysis to multiple team hierarchies and levels would be to repeat the analysis for each hierarchy and level. This would be very inefficient, however, so we develop an algorithm that performs concurrency analysis simultaneously on all hierarchies and levels.

7.2.1 The Concurrency Graph

The addition of multiple team hierarchies requires some changes to the concurrency graph used in the analysis. We describe those changes here.

7.2.1.1 Barriers

In the previous analysis, nodes corresponding to barrier operations were removed from the concurrency graph, along with all adjacent edges. With the addition of teams, barriers can be on subsets of all threads, so they only prevent code from running concurrently within the context of particular team hierarchies and levels. Thus, we can no longer completely remove them from the concurrency graph. Instead, we replace each pair of incoming and outgoing edges with a special *barrier edge* that is labeled with the set of teams on which the barrier can operate, as shown in Figure 7.10. This set of teams is computed or specified as described in §5.5.

```
a;
Ti.barrier();
teamsplit(t2(1))
    {
  b;
  Ti.barrier();
  c;
  teamsplit(t2
     (2)) {
    d;
    Ti.barrier()
       ;
    e;
  }
  f;
  partition(t7
     (2)) {
    { g;
      Ti.barrier
         ();
      h;
    }
    { i; }
  }
  j;
}
```

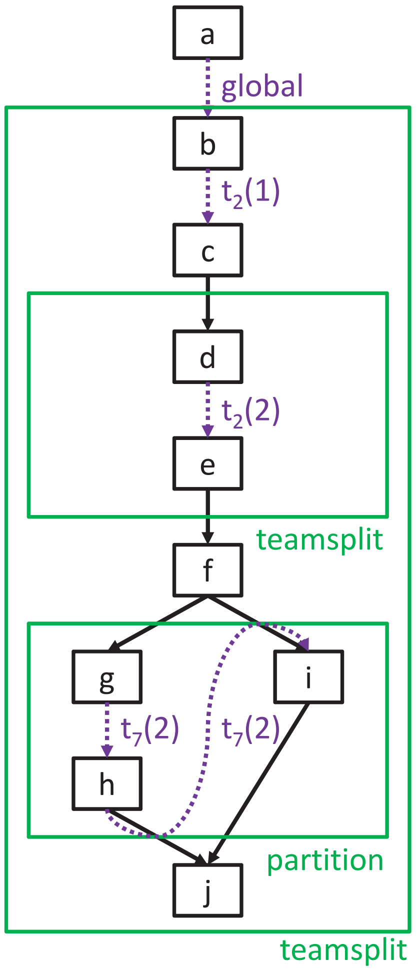


Figure 7.10: Example of a concurrency graph. Dashed edges are barrier edges.

7.2.1.2 Single Conditionals

A *single conditional* is redefined to be a conditional guarded by an expression whose value is the same on all threads *in the executing team*. As a result, all threads in the team are guaranteed to take the same branch. On the other hand, threads may take different branches of a non-single conditional, so that code in different branches may run concurrently. Since alignment of team collectives is enforced by the dynamic scheme described in §4.3, it is necessary to compute which conditionals are single. Strict and weak alignment schemes result in different sets of single conditionals.

Strict alignment requires that all threads in the executing team take the same branch of a conditional that *may execute* a collective. Such a conditional either directly contains a collective operation or calls a method that may execute a collective. The Titanium compiler already computes the set of such conditionals for the purposes of instrumenting a program to dynamically enforce collective alignment. All such conditionals are single under strict alignment, so no further computation is necessary.

Weak alignment, on the other hand, only requires that all threads in the executing team take the same branch of a conditional when any of those threads encounter a collective operation in the execution of that conditional. Statically, the compiler can only assume that a conditional is single if at least one of its two branches[6] *must execute* a collective. A non-conditional statement or expression must execute a conditional if it is a collective operation, or contains code or calls a method that must execute a collective. A conditional must execute a collective if both of its branches must execute a collective. (A conditional is single but does not must-execute a collective if exactly one of its branches must execute a collective.) It is straightforward to modify the current may-execute analysis to also compute must-execute information.

The following code demonstrates the difference between strict and weak alignment in determining the set of single conditionals.

```
1 if (a) {
2   if (b) {
3     if (c) {
4       Ti.barrier();
5     } else {
6       Ti.barrier();
7     }
8   } else {
9   }
10 } else {
11 }
```

All three conditionals are single under strict alignment, since they all contain a collective operation and may execute a collective. Under weak alignment, both branches of the conditional at line 3 must execute a collective, so it is single and must execute a collective. Only the first branch of the conditional at line 2 must execute a collective, so it is single but does not must-execute a collective. Finally, neither branch of the conditional at line 1 must execute a collective, so it is neither single nor must it execute a collective.

As before, non-single conditionals require the addition of *cross edges* in the concurrency graph that connect the end of the first branch to the beginning of the second to represent the fact that the two branches may execute concurrently. A single conditional also requires some form of cross edge, since its branches are only non-concurrent in the context of the team under which it executes. We use a *barrier cross edge* that is labeled with the possible teams under which the conditional may execute.

7.2.1.3 Partition Statements

A partition statement includes a branch on a thread's subteam. Though there may be no program path between the blocks of a partition statement, they do execute concurrently on different sub-

[6]As discussed in §7.1.1.1 *must execute* a collective, the concurrency analysis operates on an intermediate form in which all switch statements and dynamically-dispatched method calls are rewritten to if statements. The latter can be considered to always have two branches, one of which may be empty or contain nested conditionals.

teams. To represent this, cross edges must be added between blocks. Plain cross edges, however, would indicate that the blocks may run concurrently on all teams. Instead, a barrier cross edge should be added, labeled with the possible subteams that can execute the partition blocks. This would indicate that the blocks may not be concurrent in the context of those teams but may be in others.

7.2.2 Analysis

As before, the concurrency analysis consists of a series of depth-first searches (DFS), one from each node in the concurrency graph. In the original analysis, at the conclusion of each DFS, each node in the program is in one of two states: reachable or unreachable. Two nodes may be concurrent in the global team if the second is reachable from the first in the DFS starting from the first node, or if the first is reachable from the second in the DFS starting from the second node.

With the addition of teams, we are now interested in which team hierarchies and levels that two nodes may be concurrent. Two nodes are concurrent in a particular hierarchy and level if there may be two threads within the same corresponding subteam that execute the two nodes concurrently. For example, statements b and c in Figure 7.10, in the context of the teams in Figure 6.6, are concurrent at level 1 in t_m and globally (i.e. at level 0) in t_2 and t_7. On the other hand, statements a and b are not concurrent in any team hierarchy or level.

In order to represent the more complicated concurrency information, we expand the abstract state of each node in a DFS from a simple binary state to a compound state involving each set of team hierarchies.

7.2.2.1 Abstract State

The abstract state of a node consists of an $n + 1$ tuple, where n is the number of team hierarchies in the program. Due to the way in which the team lattice is constructed in §5.5.1, n is equal to the number of parent elements of the *thread local* element in a team lattice. It is also equal to the number of paths from global to thread local, since each path corresponds to a single team hierarchy.

For each element in the state tuple, the set of possible values form a lattice. The first element corresponds to the context and no-context marks in the original analysis and takes on the values $\bot = \mathit{unmarked} \sqsubset \mathit{context} \sqsubset \mathit{no\_context} = \top$. The remaining n elements correspond to the n team hierarchies. For each team hierarchy, there is a simple lattice corresponding to the levels in the team. A node is more concurrent with respect to a particular team if it is concurrent in a lower level of the team. For example, a node is more concurrent with respect to t_2 if it is concurrent in $t_2(2)$ than if it is concurrent in $t_2(1)$; if there is a thread in the same subteam at $t_2(2)$ that concurrently executes the node, then that same thread must be in the same subteam at $t_2(1)$. The reverse is not necessarily true. Thus, higher levels of the team hierarchy should be lower in the lattice. The lattice for t_2, in particular, is $\bot \sqsubset$ global $\sqsubset t_2(1) \sqsubset t_2(2) = \top$. No element for thread local is in the lattice[7], and there is an extra $\bot$ element that denotes that a node is non-concurrent

[7]We assume that a single thread cannot concurrently execute multiple expressions. Non-blocking operations violate this assumption, but we are only concerned with concurrency across threads in this analysis.

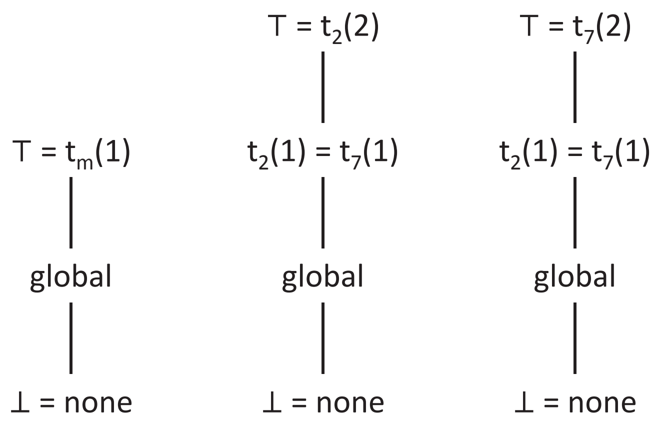


Figure 7.11: Concurrency-analysis lattices corresponding to the teams in Figure 6.6.

in t_2. Figure 7.11 shows the lattices for the set of teams from Figure 6.6. Compare it to the team lattice in Figure 5.4.

State tuples themselves form a lattice, with

$$s_1 \sqsubseteq s_2 \iff \forall i \in [0, n+1).\ s_1(i) \sqsubseteq s_2(i)$$

for any tuples s_1 and s_2. This lattice has height in $\mathrm{O}(\sum_{t \in T} height(t))$, where T is the set of all team hierarchies and $height(t)$ is the height of a particular team hierarchy t.

Discussion The abstract state defined above is only semi-hierarchical, in that it keeps track of each team hierarchy separately and does not take into account relationships between different hierarchies. This raises the question as to why we do not use a purely hierarchical abstract state. We demonstrate that such a state results in reduced precision in the analysis.

Consider the possible hierarchical state lattice shown in Figure 7.12. Using a single lattice element, how can we represent a state denoting that a node may be concurrent in $t_2(2)$ and $t_7(2)$ but not in $t_m(1)$? Such a state would arise following a barrier on $t_m(1)$, and in the semi-hierarchical abstract state described above, it would be represented as $(*, global, t_2(2), t_7(2))$. Using the purely hierarchical lattice, we could represent this state as $t_m(1)$, if a particular element denotes that the node is non-concurrent in the element's corresponding team but may be concurrent in other teams.

Now, using the proposed scheme, how can a state denoting that a node is non-concurrent in $t_m(1)$ and $t_2(2)$ but may be concurrent in $t_7(2)$ be represented? This state would arise following consecutive barriers on $t_m(1)$ and $t_2(2)$. The state $(*, global, t_2(1), t_7(2))$ would represent this in the semi-hierarchical scheme. But in the purely hierarchical scheme in which an element represents non-concurrency in the associated team, there is no single element in the lattice that corresponds to this state. The best we could do would be to use either $t_m(1)$ or $t_2(2)$ individually, losing information about non-concurrency in the other team.

Thus, a purely hierarchical scheme results in a loss of precision, so we use the semi-hierarchical scheme described above instead.

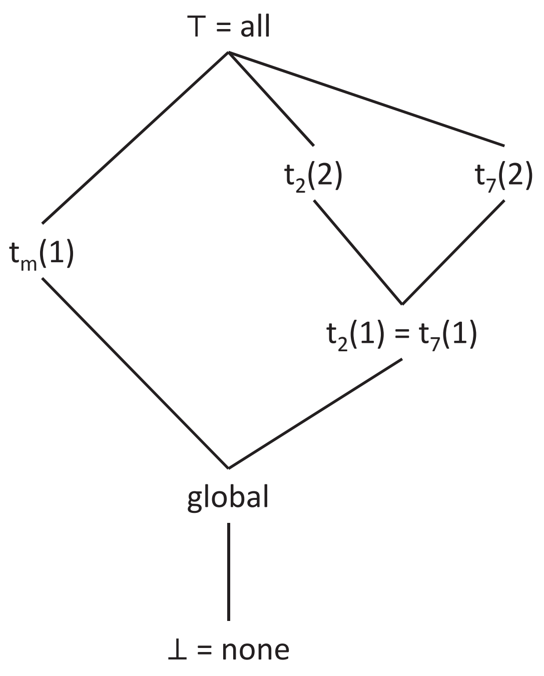


Figure 7.12: Possible hierarchical concurrency-analysis lattice corresponding to the teams in Figure 6.6.

7.2.2.2 Barrier Edges

A barrier denotes a decrease in concurrency between the code that follows it and the code that precedes it. In order to represent this in the analysis, we assign each barrier edge a static state tuple according to the team hierarchies and levels over which the barrier can execute and with the mark element assigned $\top = no\_context$.

Let us first consider a barrier edge on a single hierarchy and level $t(k)$. Then for each team hierarchy t', its state element is $child(t(k))$, where $child(t(k))$ is the child element of $t(k)$ in the lattice for t', if $t(k)$ is an element in t'. Otherwise, its state element is $\top$. As a concrete example, the five possible single hierarchy/level barrier edges for the teams in Figure 6.6 have the following states:

	$mark$	t_m	t_2	t_7
global:	(no_context,	none,	none,	none)
$t_m(1)$:	(no_context,	global,	$t_2(2)$,	$t_7(2)$)
$t_2(1)$:	(no_context,	$t_m(1)$,	global,	global)
$t_2(2)$:	(no_context,	$t_m(1)$,	$t_2(1)$,	$t_7(2)$)
$t_7(2)$:	(no_context,	$t_m(1)$,	$t_2(2)$,	$t_2(1)$)

The state of a barrier edge that operates on multiple teams is the join of the states corresponding to barrier edges on each of the individual teams. Thus, a barrier edge that can operate on $t_2(2)$ and

```
Algorithm 7.2.1.
ConcurrencyGraph($P$ : program) : graph
 1. Let $G$ be the interprocedural control-flow graph of $P$.
 2. For each conditional $C$ in $P$ {
 3.   If $C$ is not a single conditional:
 4.     Add a cross edge for $C$ in $G$.
 5. } // End for (2).
 6. For each partition statement $Q$ in $P$ {
 7.   For each pair of consecutive blocks $(b_1, b_2)$ in $Q$:
 8.     Add a barrier cross edge in $G$ from $b_1$ to $b_2$ with state as described in §7.2.2.2.
 9. } // End for (6).
10. For each barrier $B$ in $P$ {
11.   For each pair of incoming and outgoing edges $(e_1 = (u, B), e_2 = (B, v))$:
12.     Add a barrier edge $e = (u, v)$ to $G$ with state as described in §7.2.2.2.
13.   Delete $B$ and its adjacent edges from $G$.
14. } // End for (10).
15. Return $G$.
```

Figure 7.13: Algorithm 7.2.1 computes the concurrency graph of a program by inserting cross edges into its control-flow graph and replacing barrier nodes with barrier edges.

$t_7(2)$ has state

$$\begin{aligned}(\text{no\_context}, t_m(1), t_2(1), t_7(2)) \sqcup (\text{no\_context}, t_m(1), t_2(2), t_2(1)) \\ = (\text{no\_context}, t_m(1), t_2(2), t_7(2)),\end{aligned}$$

and an edge that operates on $t_2(1)$ and $t_7(2)$ has state

$$\begin{aligned}(\text{no\_context}, t_m(1), \text{global}, \text{global}) \sqcup (\text{no\_context}, t_m(1), t_2(2), t_2(1)) \\ = (\text{no\_context}, t_m(1), t_2(2), t_2(1)).\end{aligned}$$

Figure 7.13 demonstrates how the concurrency graph is constructed. Barrier edges are assigned states as discussed above.

7.2.2.3 Basic Search Algorithm

Various pieces of the concurrency analysis perform searches on concurrency graphs. These are at their core basic depth-first searches, with minor modifications. In the previous analysis, since barriers are deleted from a concurrency graph, these searches would terminate upon encountering a barrier. In the new analysis, however, barriers are not deleted, and a search path should only terminate with respect to the specific teams on which a barrier may execute.

Algorithm 7.2.2 performs a basic depth-first search over the new concurrency graph. The search has a running state, representing the set of teams in which the current node may be concurrent with the search origin. Initially, this state is set to $\top$ for all team hierarchies, denoting that the

```
Algorithm 7.2.2.
Search($v$ : vertex, $G$ : graph) :
  1. Set $state(u) \leftarrow \bot$ for all nodes $u$ in $G$.
  2. Call $DFS(v, G, \top^{n+1})$.

  3. Procedure $DFS(v$ : vertex, $G$ : graph, $q$ : search state):
  4.   Set $state(v) \leftarrow state(v) \sqcup q$.
  5.   For each edge $e = (v, u) \in G$ {
  6.     Let $q' \leftarrow q$.
  7.     If $e$ is a barrier edge:
  8.       Set $q' \leftarrow q' \sqcap state(e)$.
  9.     If $(state(u) \sqcup q') \sqsupset state(u)$:
 10.       Call $DFS(u, G, q')$.
 11.   } // End for (5).
```

Figure 7.14: Algorithm 7.2.2 performs a depth-first search on a concurrency graph.

start node is concurrent with itself in all teams. Each node starts out with a state of $\bot$. When a new node is visited, its state is set to the join of its old state and the current search state. For example, if a node is adjacent to the start with no intervening barrier edge, then its state is also set to $\top$, and it is concurrent with the start node in all teams. On the other hand, upon encountering a barrier edge, the new search state is the meet of the old search state and the state of the barrier edge, denoting that subsequent nodes are not concurrent in the team on which the barrier executes, in the case of a barrier that operates on only one team. A node is only revisited if its state changes, and a node is concurrent with the start node in a particular team hierarchy and level $t(k)$ if its final state element for hierarchy t subsumes $t(k)$, i.e. is a subset of $t(k)$.

In order to demonstrate correctness of Algorithm 7.2.2, we must show that on a single team hierarchy and level $t(k)$, it marks exactly the same set of nodes concurrent in $t(k)$ as a normal depth-first search on a concurrency graph constructed specifically for $t(k)$, with barriers that operate only on $t(k)$ or a superset of $t(k)$ removed from the graph.

Theorem 7.2.3. *In a search starting from node v, Algorithm 7.2.2 marks the same set of nodes in graph G concurrent in $t(k)$ as a normal depth-first search on G', a graph equivalent to G but with all barrier edges that operate only on a superset of $t(k)$ removed.*

Proof. Consider an arbitrary path R in G taken by the search in Algorithm 7.2.2. The search state at any point in the path is the meet of $\top$ and the states of all barrier edges along the path. Prior to encountering a barrier edge that operates only on a superset of $t(k)$, the search state always has a subset of $t(k)$ as its element for hierarchy t, since each barrier edge that does not operate solely on a superset of $t(k)$ has a subset of $t(k)$ as its state element for t. Thus, all encountered nodes are marked as concurrent in $t(k)$. In the standard DFS on G', no encountered barrier edges have been removed, since they do not operate solely on a superset of $t(k)$, so it will also follow the same path and mark every node as concurrent.

Upon encountering a barrier that operates only on a superset of $t(k)$, the standard DFS on G' stops, since the corresponding edge is deleted from G', marking no more nodes along R. Algorithm 7.2.2, on the other hand, sets the search state to the meet of the previous state and that of the barrier edge. Since the barrier operates only on a superset of $t(k)$, its state element for t is the parent team of the lowest subteam on which the barrier operates, which is a strict superset of $t(k)$. Thus, the new search state also has a strict superset of $t(k)$ as its element for t. Since a new search state is always the meet of the previous state and some other state, the search state in R after the barrier will always have a strict superset of $t(k)$ as its element for t, so that none of the remaining nodes in R will be marked as concurrent in $t(k)$.

Since both searches mark the same set of nodes as concurrent in $t(k)$ before and after encountering a barrier on a superset of $t(k)$, they mark the same set of nodes as concurrent in $t(k)$. $\square$

A basic depth-first search can only visit each node $O(\sum_{t \in T} height(t))$ times, since a node is only visited when its state changes. Thus, the search runs in $O(n \cdot \sum_{t \in T} height(t))$, where n is the number of nodes in G[8].

7.2.2.4 Bypass Edges

As described in §7.1.3.1, a *bypass edge* connects a call node and a return node, bypassing the body of the invoked method. In the hierarchical analysis, bypass edges must be modified to be barrier edges, denoting which barriers must be encountered between the start and end of a method. Each method is now assigned a state tuple, initially set to $\bot$. The bypass computation algorithm continually iterates over all methods as before, until no changes are made. However, in each iteration, the intraprocedural searches now carry a state, in each iteration initially set to $\top$. Upon encountering a barrier edge, the new search state is the meet of the old search state and the tuple for the barrier edge. Similarly, upon encountering a method call, the new search state is the meet of the old search state and that of the target method. Upon encountering any other node, the state of that node is set to the join of its previous state and the search state. If the node's state changes, it is revisited in the context of the search state. exit is reached, the method's state is updated to be the join of its old state and the search state.

When an iteration of the bypass computation algorithm makes no changes to any method's state, the algorithm completes, and a bypass barrier edge is added for each method call with the target method's final state.

At most $k + 1$ iterations are required, where k is the number of methods in the program, since at least one method is marked in all but the last iteration. An iteration visits each of the n nodes in graph G at most $O(\sum_{t \in T} height(t))$ times, so that it takes time in $O(n \cdot \sum_{t \in T} height(t))$. Thus, Algorithm 7.2.4 takes time in $O(kn \cdot \sum_{t \in T} height(t))$.

```
Algorithm 7.2.4.
ComputeBypasses($P$ : program, $G_1, \ldots, G_k$ : intraprocedural concurrency graph) :
 1. Let $change \leftarrow true$.
 2. Set $state(f) \leftarrow \bot$ for all methods $f$ in $P$.
 3. While $change = true$ {
 4.   Set $change \leftarrow false$.
 5.   Set $state(u) \leftarrow \bot$ for all nodes $u$ in $G_1, \ldots, G_k$.
 6.   For each method $f$ in $P$ {
 7.     Call $CanReach(entry(f), exit(f), G_f, \top)$.
 8.     If $state(exit(f)) \sqsupset state(f)$ {
 9.       Set $state(f) \leftarrow state(exit(f))$.
10.       Set $change \leftarrow true$.
11.     } // End if (8).
12.   } // End for (6).
13. } // End while (3).

14. Procedure $CanReach(u, v$ : vertex, $G$ : graph, $q$ : search state):
15.   Set $state(u) \leftarrow state(u) \sqcup q$.
16.   If $u = v$:
17.     Return.
18.   Else If $u$ is a method call to function $g$:
19.     Set $q \leftarrow q \sqcap state(g)$.
20.   For each edge $e = (u, w) \in G$ {
21.     Let $q' \leftarrow q$.
22.     If $e$ is a barrier edge:
23.       Set $q' \leftarrow q' \sqcap state(e)$.
24.     If $(state(w) \sqcup q') \sqsupset state(w)$:
25.       Call $CanReach(w, v, G, q')$:
26.   } // End for (20).
```

Figure 7.15: Algorithm 7.2.4 uses each method's intraprocedural concurrency graph to determine under which teams its exit is reachable from its entry.

```
Algorithm 7.2.5.
FeasibleSearch($v$ : vertex, $G$ : graph) :
 1. Set $state(u) \leftarrow \bot$ for all nodes $u$ in $G$.
 2. Let $s \leftarrow \emptyset$.
 3. Call $FeasibleDFS(v, G, s, \top^{n+1})$.
 4. Set $concur(\{v,u\}) \leftarrow concur(\{v,u\}) \sqcup state(u)$ for all nodes $u$ in $G$.

 5. Procedure $FeasibleDFS(v$ : vertex, $G$ : graph, $s$ : stack, $q$ : search state):
 6.   Set $state(v) \leftarrow state(v) \sqcup q$.
 7.   For each edge $e = (v,u) \in G$ {
 8.     Let $s' \leftarrow s$.
 9.     Let $q' \leftarrow q$.
10.     If $label(e)$ is a close symbol and $s' \neq \emptyset$ {
11.       Let $o \leftarrow pop(s')$.
12.       If $label(e)$ does not match $o$:
13.         Skip to next iteration of 7.
14.       Else if $s' = \emptyset$:
15.         Set $q' \leftarrow (no\_context) + q'(1:n)$.
16.     } // End if (10).
17.     Else if $label(e)$ is an open symbol {
18.       If $s' = \emptyset$:
19.         Set $q' \leftarrow (context) + q'(1:n)$.
20.       Push $label(e)$ onto $s'$.
21.     } // End else (17).
22.     Else if $e$ is a barrier edge:
23.       Set $q' \leftarrow q' \sqcap state(e)$.
24.     If $(state(u) \sqcup q') \sqsupset state(u)$:
25.       Call $FeasibleDFS(u, G, s', q')$.
26.   } // End for (7).
```

Figure 7.16: Algorithm 7.2.5 computes the set of teams under which each node is reachable from the start node through a feasible path.

7.2.2.5 Feasible-Search Algorithm

We modify the feasible-search algorithm to incorporate the new abstract states. The search keeps track of a *current state* that consists of a single state tuple. Initially, this is set to a tuple with the no context mark and $\top$ for each team hierarchy. This state is modified in the following cases:

1. When the stack is empty and an open parenthesis symbol is encountered, the mark is set to context.
2. When a parenthesis is popped off the stack, if the stack becomes empty, then the mark is set to no context.
3. When a barrier edge is encountered, the new state is the meet of the old state and the state assigned to the barrier edge.

When visiting a node, the search sets the node's new state to be the join of the current search state and the node's previous state. The node is reexamined only if its state changes.

Algorithm 7.2.5 visits each node at most $\mathrm{O}(\sum_{t \in T} height(t))$ times, since it only visits a node when its state changes. Thus, it runs in time $\mathrm{O}(n \cdot \sum_{t \in T} height(t))$, where n is the number of nodes in G.

7.2.2.6 Context Marks

The previous analysis used separate state elements to denote the context and no-context marks, whereas in §7.2.2.1, we have combined them into a single element in which no-context subsumes context. This is only valid if all paths that are feasible in a particular initial context are also feasible in no initial context.

Theorem 7.2.6. *Any path R in graph G that is feasible in an initial context S is also feasible in an initially empty context.*

Proof. In order for R to be feasible, the labels encountered along R must match. In particular, any open parenthesis $(_\alpha$ in R must either be closed by a matching close parenthesis $)_\alpha$ or not closed at all in R. Excluding matching sets of open and close parentheses in R, the remaining labels in R consist of m unmatched close parentheses followed by n unmatched open parentheses, $m, n \geq 0$. In order for R to be feasible in an initial context S' consisting of k parentheses, the first $min(m, k)$ unmatched close parentheses in R must match those in S'. This holds for $S' = S$, since R is feasible in S. It also holds for $S' = \emptyset$, since k is then 0 and none of the m unmatched close parentheses in R need be matched by the context in order for R to be feasible in S'. Thus, R is feasible an an initially empty context. □

Thus, it is valid to use only a single state element to keep track of the context and no-context marks.

[8] The concurrency graph G is very sparse, with only $\mathrm{O}(n)$ edges, so there is no explicit dependency on the number of edges in the running time.

```
Algorithm 7.2.7.
FeasibleConcurrentExpressions($P$ : program) :
 1. Let $G \leftarrow$ ConcurrencyGraph($P$) [Algorithm 7.2.1].
 2. For each method $f$ in $P$ {
 3.   Construct the intraprocedural concurrency graph $G_f$ of $f$.
 4.   For each barrier $B$ in $f$ {
 5.     For each pair of edges $(e_1 = (u, B), e_2 = (B, v))$ in $G_f$:
 6.       Add a barrier edge $e = (u, v)$ to $G_f$ with state as described in §7.2.2.2.
 7.     Delete $B$ and its adjacent edges from $G_f$.
 8.   } // End for (4).
 9. } // End for (2).
10. Call ComputeBypasses($P$, $G_1, \ldots, G_k$) [Algorithm 7.2.4].
11. For each method call and return pair $c, r$ in $P$ with target $f$:
12.   Add a barrier edge $(c, r)$ to $G$ with state $state(f)$.
13. Let $concur(\{a, b\}) \leftarrow \perp$ for each pair of nodes $a, b$ in $G$.
14. For each node $a$ in $G$:
15.   Call FeasibleSearch($a$, $G$) [Algorithm 7.2.5].
```

Figure 7.17: Algorithm 7.2.7 computes the set of teams under which any pair of expressions can feasibly and concurrently execute in a given program.

7.2.2.7 Concurrent Expressions

The complete algorithm to compute concurrent expressions is shown in Figure 7.17. It begins by using Algorithm 7.2.1 to construct the interprocedural concurrency graph. It then constructs corresponding intraprocedural graphs for each method and calls Algorithm 7.2.4 to compute bypass edges, adding them to the concurrency graph. Finally, it calls Algorithm 7.2.5 starting from each node in the graph. At the end of the algorithm, $concur(\{a, b\})$ is a state tuple describing at which level expressions a and b may be concurrent in each team hierarchy.

Algorithm 7.2.7 takes time in $\mathrm{O}(n)$ to construct the concurrency graphs, where n is the number of statements and expressions in the program. It calls Algorithm 7.2.4, which takes time in $\mathrm{O}(kn \cdot \sum_{t \in T} height(t))$, where k is the number of methods in the program. It calls Algorithm 7.2.5 $\mathrm{O}(n)$ times, taking time in $\mathrm{O}(n^2 \cdot \sum_{t \in T} height(t))$. Since k is in $\mathrm{O}(n)$, the total running time for Algorithm 7.2.7 is in $\mathrm{O}(n^2 \cdot \sum_{t \in T} height(t))$.

The theoretical running time for this analysis is exactly the same as for running the original concurrency analysis separately for each team hierarchy and level. In practice, however, we expect the actual running time of the new analysis to be far lower than repeating the old analysis, since the new analysis performs simultaneous analysis on all teams.

Chapter 8

Evaluation

We now turn our attention to evaluating the recursive single program, multiple data (RSPMD) extensions in §3, the alignment scheme we proposed in §4, and the pointer and concurrency analyses we described in §6 and §7. We start by examining four separate applications, writing or rewriting them in the RSPMD model, evaluating both expressiveness of the model and performance. We then test implementations of the dynamic alignment scheme for collectives, providing performance results for both raw collectives and complete application benchmarks. Finally, we evaluate pointer analysis using locality and sharing inference and then combine it with concurrency analysis for race detection and enforcement of sequential consistency.

8.1 Application Case Studies

In order to guide the design of the language constructs described in §3 and evaluate their effectiveness, we examined four application benchmarks to determine how they can benefit from the new hierarchical team constructs. In this section, we present case studies of the four applications, conjugate gradient, parallel sort, particle in cell, and stencil.

8.1.1 Test Platforms

We tested application performance on three machines, a Cray XT4, a Cray XE6, and an IBM iDataPlex, all located at the National Energy Research Scientific Computing Center (NERSC) [75] at the Lawrence Berkeley National Laboratory (Berkeley Lab) [62]. The Cray XT4, named *Franklin*, was a cluster of quad-core AMD Budapest 2.3 GHz processors, with one quad-core processor per node and a SeaStar-2 interconnect. The Cray XE6, called *Hopper*, consists of two twelve-core AMD MagnyCours 2.1 GHz processors per node, each of which consists of two six-core dies. Each die is referred to as a *non-uniform memory access (NUMA) node*, since each die has fast access to its own memory banks but slower access to the other banks. The XE6 uses a custom Gemini interconnect for communication. On both Cray machines, we used the MPI conduit of GASNet for communication. The IBM iDataPlex system, known as *Carver*, is a cluster of eight-core, 2.67 GHz Intel Nehalem processors connected by a 4X QDR InfiniBand network. We used the native Infiniband conduit of GASNet for communication on this system. The largest job size on the iDataPlex is limited to 64 nodes by system policy, though in practice, memory considerations limited us to 32 nodes for most benchmarks and prevented larger problem sizes from being run.

In most of the benchmark applications, we focused on optimizing distributed performance. As a result, we used performance on a single shared-memory node or NUMA node as the baseline for our experiments. Optimizing execution solely on shared-memory multicores is beyond the scope of this thesis. However, all benchmarks were run with runtime error checking disabled, including those for dynamic alignment, to avoid any unnecessary overhead.

8.1.2 Conjugate Gradient

The conjugate gradient (CG) application is one of the NAS parallel benchmarks [9]. It iteratively determines the minimum eigenvalue of a sparse, symmetric, positive-definite matrix. The matrix is divided in both dimensions, and each thread receives a contiguous block of the matrix, with threads placed in row-major order. The application performs numerous sparse matrix-vector multiplications. Consider a blocked matrix-vector multiplication, as illustrated previously in Figure 3.3. Each element in the source vector must be distributed to the threads that own a portion of the corresponding matrix column. Each element in the destination vector is computed using a reduction across the threads that own a portion of the corresponding matrix row. Since the algorithm is iterative, each segment of the result vector must be distributed to those threads that require it in the next iteration, in which the previous result becomes the new source vector.

8.1.2.1 Original Implementation

Prior to our language extensions, Titanium only supported collectives over all threads in a program. Thus, the original Titanium implementation of CG [29] required hand-written reductions over subsets of threads. These reductions required extensive development effort to implement, test, and optimize. As we discuss in §8.4.1.1, one version of the hand-written code also contains a significant bug; we use the bug-free version here.

Figure 8.1 shows the execution of the original matrix-vector multiplication in a single iteration of CG. The implementation performs an all-to-all reduction on each row, so that threads 0 to 3 receive the first half of the result vector and threads 4 to 7 receive the second half. However, in the next iteration, threads 0 and 4 require the first quarter of that result vector, threads 1 and 5 the next quarter, and so on. Thus, a partial transpose is required over the threads to transfer data to those threads that need it. In this example, threads 0, 1, 6, and 7 already have their required data, but threads 2 and 4 need to swap their results, as do threads 3 and 5.

8.1.2.2 Row Teams

The first step in modifying CG to use teams was to replace the hand-written all-to-all reductions with built-in reductions on teams. The code already computes the row and column number of each

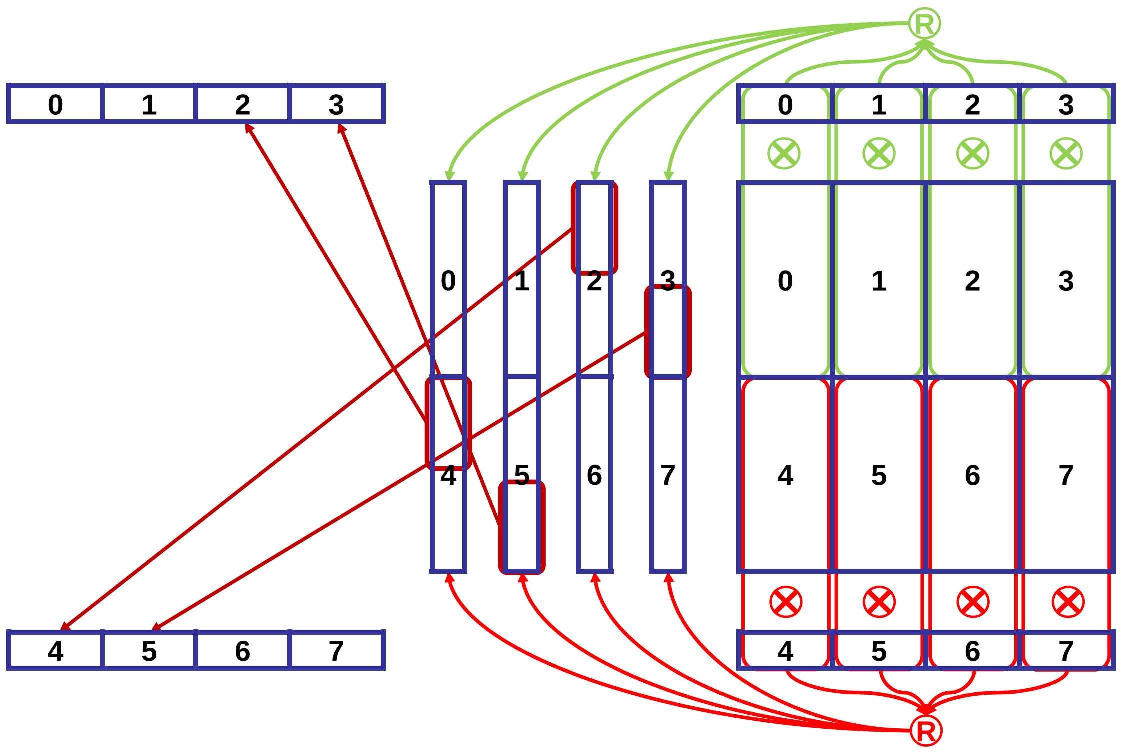


Figure 8.1: A single iteration of matrix-vector multiplication in the original and row team versions of CG.

thread, which we use to divide the threads into row teams with a call to `splitTeamAll`(). Then in each matrix-vector multiplication, a single library call is all that is necessary to perform a reduction across each row team, as shown in the code below.

```
Team rowTeam = new Team();
public void initialize() {
  rowTeam.splitTeamAll(rowPos, colPos);
  rowTeam.initialize(false);
  ...
}
public void multiply(Vector in, Vector out) {
  ...
  teamsplit(rowTeam) {
    // Reduce results of this row to all threads in the row.
    Reduce.add(combinedResults, myResults);
  }
  ...
}
```

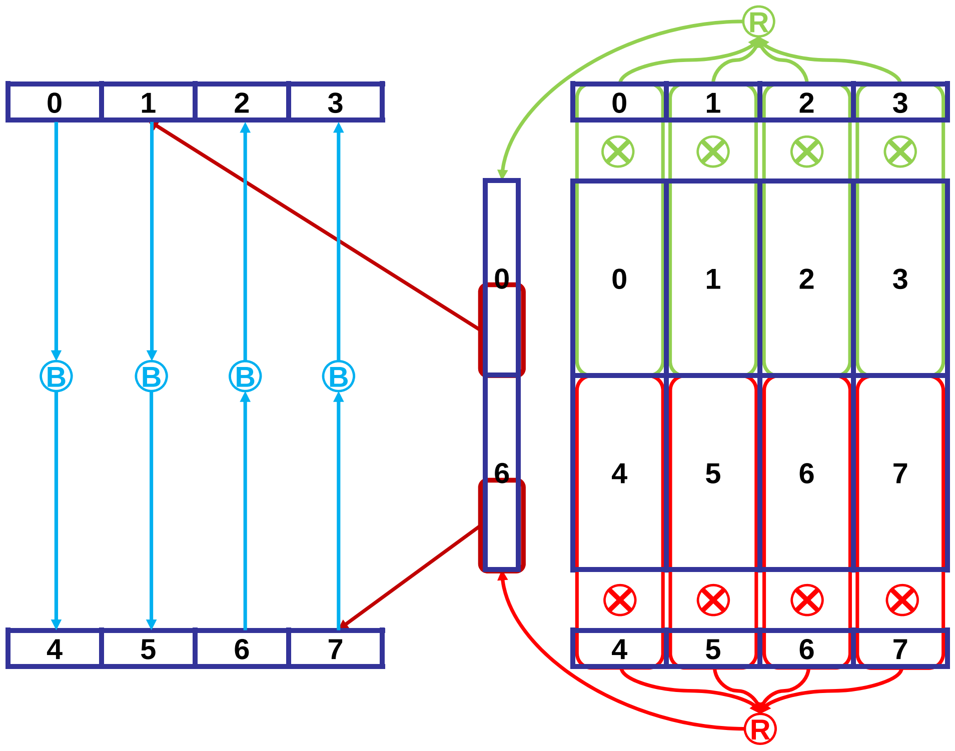


Figure 8.2: A single iteration of matrix-vector multiplication in the column team version of CG.

8.1.2.3 Column Teams

The above implementations perform unnecessary communication, since the results of the all-to-all reductions end up on every thread in a row team before being transposed across columns. In order to further optimize the code, we replaced the all-to-all reductions with all-to-one reductions, as shown in Figure 8.2. In this example, only thread 0 receives the result of the reduction in the first row team, while thread 6 receives the result of the second team. Thread 1 then copies the second half of thread 0's result, which is required by the second column of threads, and thread 7 similarly copies from thread 6. Finally, a broadcast is done in each column, transferring data from thread 0 to 4, thread 1 to 5, thread 6 to 2, and thread 7 to 3.

As before, a team for each row is required to perform the reductions. In addition, a team for each column is required to perform the broadcasts. Lastly, we construct additional teams to synchronize the source and destination threads of the copies between the reductions and broadcasts. The code below demonstrates these operations.

```
teamsplit(rowTeam) {
  // Reduce results of this row to a single target thread.
  Reduce.add(combinedResults, myResults, rowTarget);
}
teamsplit(copyTeam) {
  if (copySync) {
    // Synchronization is required if the source and destination
    // threads differ in the copy below.
    Ti.barrier();
  }
}
if (reduceCopy) {
  // This thread is responsible for copying data from the given
  // source.
  myOut.copy(allResults[reduceSource]);
}
teamsplit(columnTeam) {
  // Broadcast data from the source for this column.
  myOut.vbroadcast(columnSource);
}
```

8.1.2.4 Performance

The CG application demonstrates the importance of teams for collective operations among subsets of threads. It also illustrates the need for multiple team hierarchies and for separating team creation from usage, as the cost for creating teams is amortized over all iterations of the algorithm.

Figures 8.3, 8.4, and 8.5 compare the performance of the three versions of CG on a Cray XT4, a Cray XE6, and an IBM iDataPlex. We show strong scaling (fixed problem size) results using two problem sizes, Class B for one to 128 threads and Class D for 128 to 1024 threads. (Both axes in the figures use logarithmic scale, so ideal scaling would appear as a line on the graphs.) As expected, the replacement of hand-written reductions with optimized team reductions in the row teams version improves performance over the original version. The communication optimizations resulting from the addition of column teams further improves performance, achieving speedups over the original code of 2.1x for Class B at 128 threads and 1.6x for Class D at 1024 threads on the XT4. The XE6 shows similar speedups of 1.6x and 1.5x for the same problem sizes and thread counts. On the IBM iDataPlex, Class B only scales until 64 threads, at which point the column team version is 2.1x as fast as the original code. Class D achieves a speedup of 1.6x at 256 threads, at which point the original version stops scaling, and 2.7x at 512 threads.

As for parallel scaling, the column team version achieves a speedup of 26x for Class B on 128 threads over the sequential version on the XT4, 44x on 128 threads on the XE6, and 25x on 64 threads on the iDataPlex. It should be no surprise that the implementation ceases to scale for Class B, since communication time dominates computation time for this problem size at higher numbers of threads. For Class D, we achieve a speedup of about 4x on 1024 threads over 128 threads on both Cray machines and 2x on 512 threads on the IBM machine.

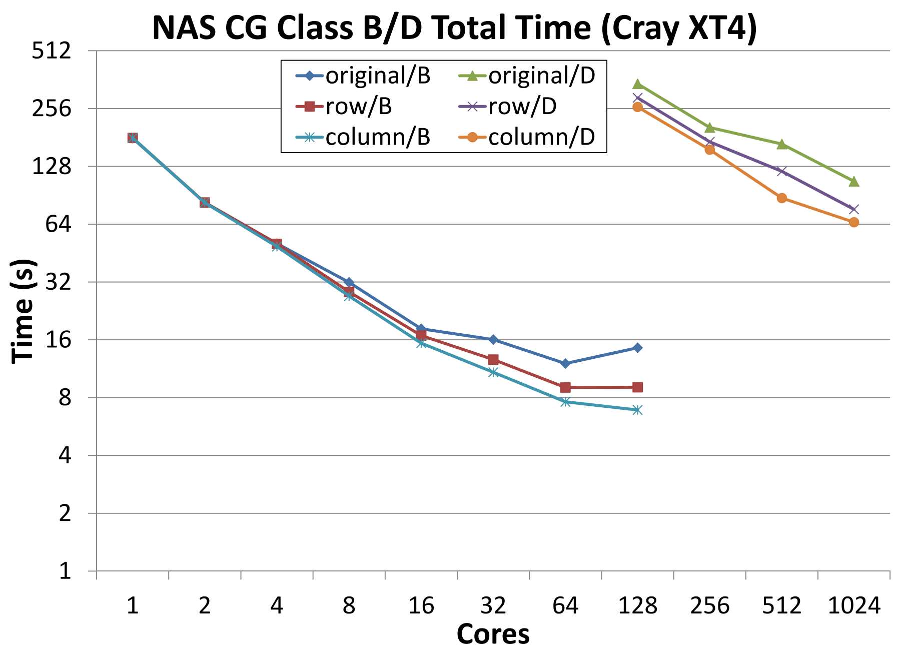


Figure 8.3: Strong scaling performance of conjugate gradient on a Cray XT4.

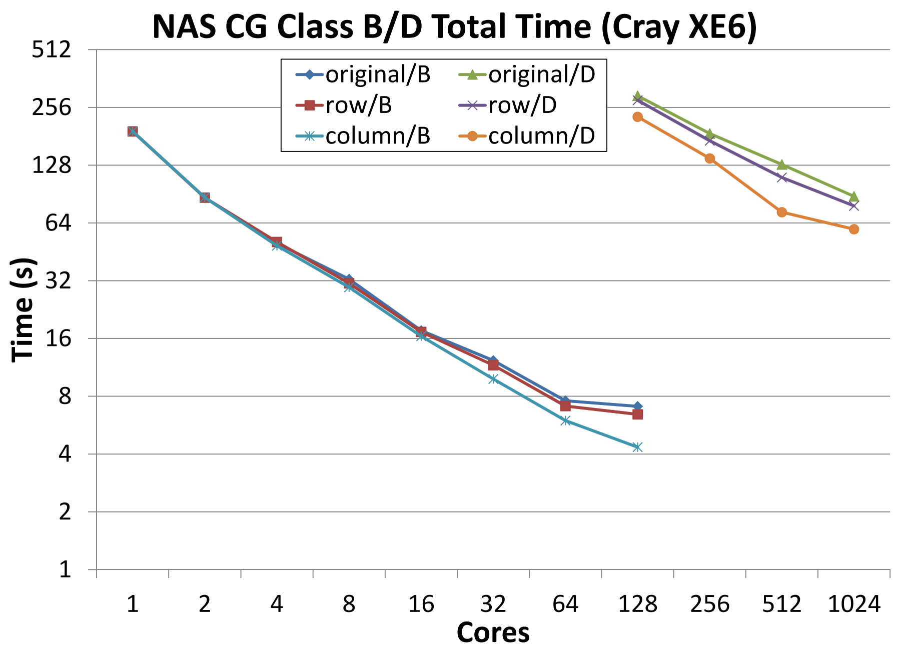


Figure 8.4: Strong scaling performance of conjugate gradient on a Cray XE6.

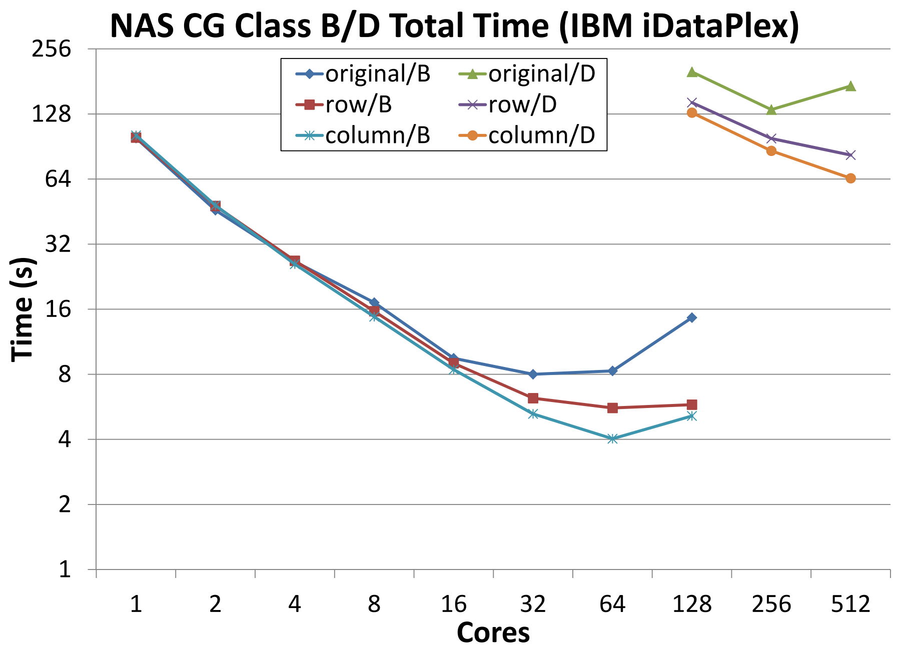


Figure 8.5: Strong scaling performance of conjugate gradient on an IBM iDataPlex.

8.1.2.5 Shared-Memory Optimizations

In addition to rewriting the CG code to use team collectives, we wrote an experimental version that explores various shared-memory optimizations. In particular, if threads that share memory are placed in the same column, these threads can share their portion of the input vector. For example, in Figure 8.2, if threads 0 and 4 share memory, they can use the same result vector rather than requiring it to be transferred from 0 to 4. There may be more threads in a column than in a shared-memory node, so column transfers may not be completely eliminated. However, they would only require a single thread from each node in a column rather than all threads, reducing the amount of communication.

In implementing this optimization, we first had to divide the matrix among threads in column-major order, since the Titanium runtime assigns contiguous blocks of thread IDs to each shared-memory node. Figure 8.6 shows the required layout. Unfortunately, the assumption of row-major order is pervasive in the code, so directly making this change proved difficult. Instead, we constructed a new global team that merely rearranged thread IDs to produce the desired ordering. We then called the unmodified CG code in the context of this team, as follows:

```
public static void main(String[] args) {
  ... // Compute number of rows and columns.
  int id = Ti.thisProc();
  // Compute ID in column-major team.
  id = id / numRows + numCols * (id % numRows);
```

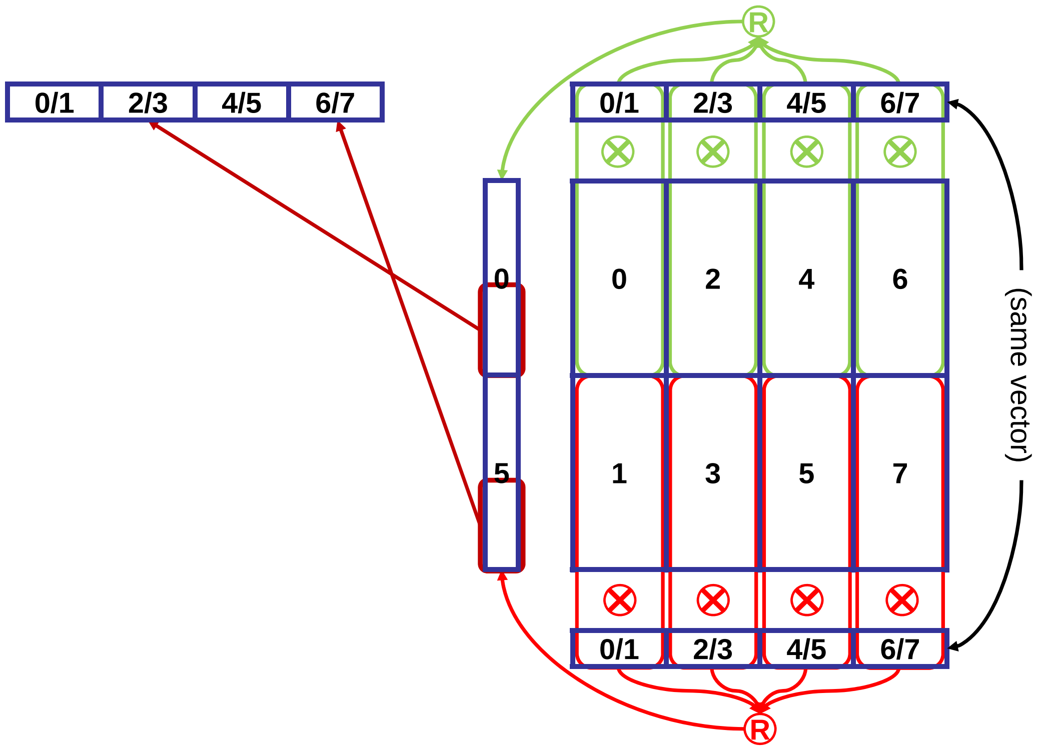


Figure 8.6: A single iteration of matrix-vector multiplication in the shared-memory version of CG, with threads arranged in column-major order.

```
  // Construct column-major team.
  Team flipTeam = new Team();
  flipTeam.splitTeamAll(0, id);
  flipTeam.initialize(false);
  // Execute unmodified CG code in the context of column-major team.
  teamsplit(flipTeam) {
    oldMain(args);
  }
}
public static void oldMain(String[] args) {
  ...
}
```

We then had to divide each column team into subteams of shared-memory threads to allow for synchronization on their shared piece of the input vector. We also had to construct teams with one thread from each node in a column in order to perform the broadcasts. The code below constructs both sets of teams.

```
  teamsplit(columnTeam) {
    // Divide column team into threads that share memory.
    colSMPTeam = new Team();
    colSMPTeam.splitTeamSharedMem(id);
    // Divide column team into subteams with one thread from each
    // node.
    colSliceTeam = colSMPTeam.makeTransposeTeam();
    colSMPTeam.initialize(false);
    colSliceTeam.initialize(false);
  }
```

Unfortunately, we determined that in most cases, the synchronization overhead from sharing vectors was greater than the time saved by reduced communication in the current version of the code. Despite the disappointing performance results, this exercise demonstrates the benefits of the new team constructs in exploring optimizations, and we plan to investigate whether or not further optimizations can make this version of the code more efficient.

8.1.3 Parallel Sort

The second application we examined was a sorting library that sorts 32-bit integers in parallel. We postulated that the most efficient implementation would be a hierarchical distributed sort that uses a communication-optimized algorithm between threads that do not share memory but otherwise takes advantage of shared memory. We started with two existing implementations, the sequential quicksort from the `java.util.Arrays` class in the Java 1.4 library and a distributed sample sort written in Titanium by Kar Ming Tang.

8.1.3.1 Overview of Sample Sort

In the sample sort algorithm [44], data is initially randomly and evenly distributed among all processors. At the end of the algorithm, the elements satisfy two properties: (1) all elements on an individual processor are in sorted order and (2) all elements on processor i are less than any element on processor $i + 1$. The algorithm accomplishes this by first randomly sampling the elements on each of the n processors, sending them to processor 0. Processor 0 sorts the samples, determining $n - 1$ pivots that divide the elements into n approximately equal sets. Each processor then uses these pivots to divide its elements into n buckets, which are then exchanged among all processors to satisfy property (2) above. Then each processor sequentially sorts its resulting elements to satisfy property (1), and the algorithm terminates.

The bucket exchange operation requires $n(n - 1)$ messages, since each processor must send $n - 1$ buckets to a different processor. On a cluster of multiprocessors, however, we speculated that it would be more efficient to aggregate communication by using only a single bucket for each node, so that $m(m - 1)$ messages would be required for m nodes. Each node's data could then be sorted in parallel by the processors on that node.

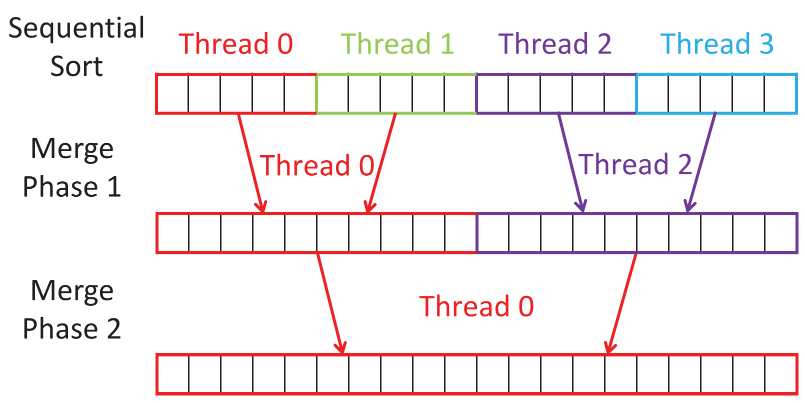


Figure 8.7: Shared-memory sorting algorithm on four threads.

8.1.3.2 Shared-Memory Sort

The original implementation of sample sort treats all threads as if they do not share memory. In order to remedy this, we first wrote a new sort implementation that assumes that all threads share memory, relying heavily on the new team constructs in writing the code. This sort does not use sample sort but a combination of merge sort and quicksort.

The shared-memory algorithm starts with a single, contiguous array of integers. This array is divided equally among all threads, each of which calls the sequential quicksort to sort its elements in-place. The separately sorted subsets are then merged in parallel in multiple phases, with the number of participating threads halved in each stage. Figure 8.7 illustrates this process on four threads.

The recursive nature of the sorting can be easily represented with a team hierarchy consisting of a binary tree, in which each node contains half the threads as its parent. The following code constructs such a hierarchy, using the **splitTeam**() library function to divide a team in half.

```
static void divideTeam(Team t) {
  if (t.size() > 1) {
    t.splitTeam(2);
    divideTeam(t.child(0));
    divideTeam(t.child(1));
  }
}
```

Then each thread walks down to the bottom of the team hierarchy, sequentially sorts its elements, and then walks back up the hierarchy to perform the merges. In each internal team node, a single thread merges the results of its two child nodes before execution proceeds to the next level in the hierarchy. The following code performs the entire algorithm, and Figure 8.8 illustrates the process on six threads.

```
static single void sortAndMerge(Team t) {
  if (Ti.numProcs() == 1) {
```

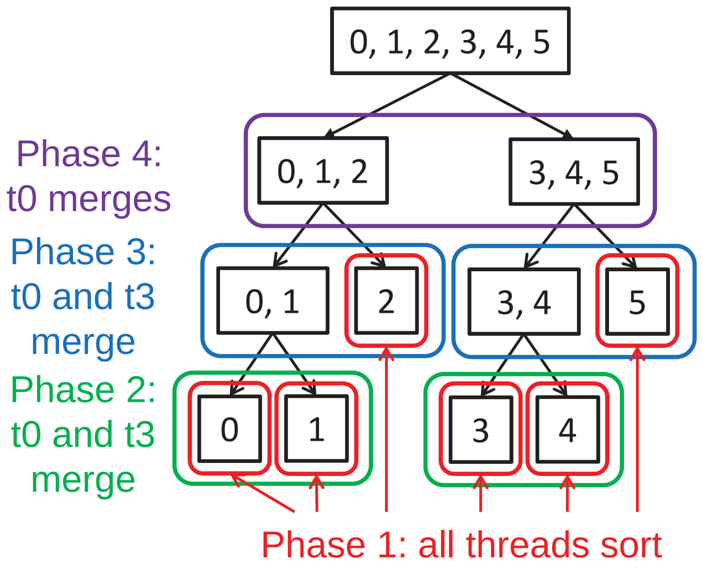


Figure 8.8: Shared-memory team hierarchy and execution on six threads.

```
    allRes[myProc] = SeqSort.sort(myData);
  } else {
    teamsplit(t) {
      sortAndMerge(Ti.currentTeam());
    }
    Ti.barrier(); // ensure prior work complete
    if (Ti.thisProc() == 0) {
      int otherProc = myProc + t.child(0).size();
      int[1d] myRes = allRes[myProc];
      int[1d] otherRes = allRes[otherProc];
      int[1d] newRes = target(t.depth(), myRes, otherRes);
      allRes[myProc] = merge(myRes, otherRes, newRes);
    }
  }
}
```

As illustrated in the code above, the shared-memory sorting algorithm is very simple to implement using the new team constructs. The entire implementation is only about 90 lines of code (not including test code and the sequential quicksort) and took just two hours to write and test.

8.1.3.3 Distributed Sort

As an initial hierarchical, distributed sort implementation, we started with an unoptimized sample-sort implementation written by Kar Ming Tang in 1999. Our initial implementation assigns a single thread from each node to participate in the sample sort, so that the number of messages is

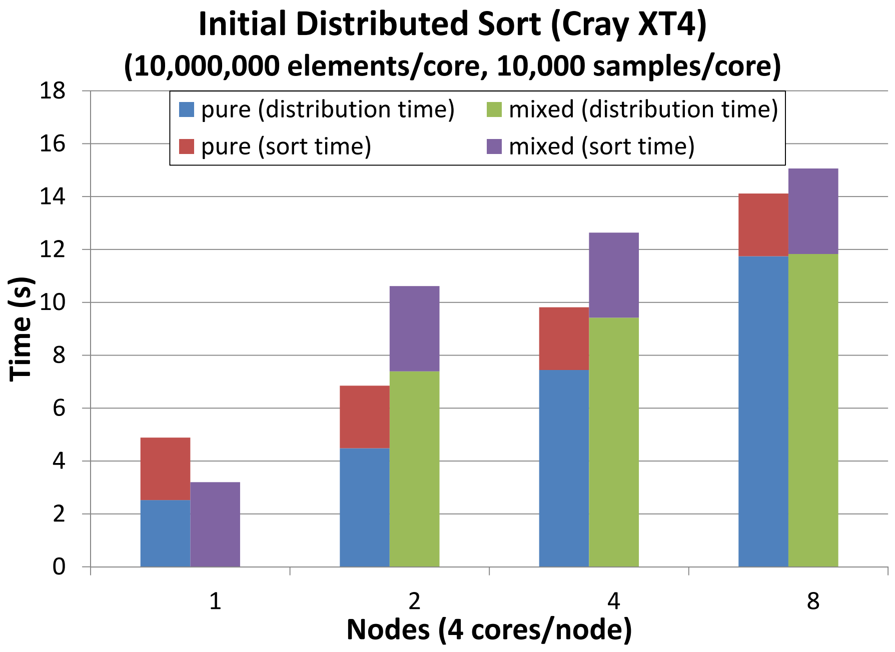


Figure 8.9: Performance of initial distributed sort on a Cray XT4 with a constant number of elements per thread.

minimized as described previously. Then each node performs the shared-memory sort described above. The entire code to accomplish this is as follows:

```
Team team = Ti.defaultTeam();
team.initialize(false);
Team oTeam = team.makeTransposeTeam();
oTeam.initialize(false);
partition(oTeam) {
  { sampleSort(); }
}
teamsplit(team) {
  keys = SMPSort.parallelSort(keys);
}
```

Again, the new team constructs make this algorithm trivial to implement, requiring only 10 lines of code and 5 minutes of development time. The code calls `Ti.defaultTeam()` to obtain a team in which threads are divided according to which threads share memory. It then uses the `makeTransposeTeam()` library call to construct a transpose team in which each subteam contains one thread from each node. The `partition` construct is then used to perform the sample sort on one of those subteams, after which the node teams execute the shared-memory sort.

This example illustrates the value of the composability features of the team extensions. As far as the code in `sampleSort()` is concerned, its entire world consists of just a single thread from

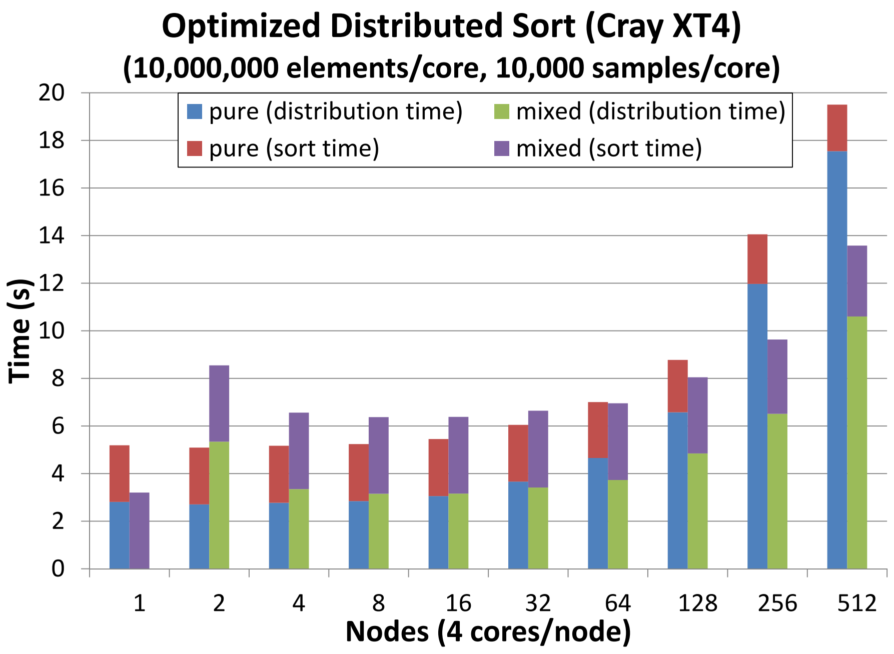


Figure 8.10: Performance of optimized distributed sort on a Cray XT4 with a constant number of elements per thread.

each node. The only change required was to remove the call to sequential sort after the sampling and distribution. Similarly, as far as the shared-memory sort is concerned, its entire world consists of the threads on a single node. No changes were required, and the team hierarchy constructed in the shared-memory sort composes cleanly with the hierarchy used here.

Figure 8.9 illustrates the performance of this initial implementation on a Cray XT4 compared to a pure sample sort, with a constant number of elements per thread. Sorting takes longer in the mixed, hierarchical version, since it is done in parallel, with threads idling in the merge phases. However, the distribution portion of the algorithm takes approximately the same time in both versions at 8 nodes. We speculated that the distribution time in the mixed version would take less time than the pure version at larger numbers of nodes.

The initial implementation does not scale beyond 8 nodes in either the pure sample sort or the mixed, hierarchical version, so we completely reimplemented the sample sort. We omit the details here, but the new version uses all threads to help with sampling and distribution rather than a single thread per node in the mixed version of the code. Communication, however, is still aggregated at the node level to minimize the number of messages required. Figures 8.10 and 8.11 compare the performance of pure sample sort and mixed, hierarchical sample and shared-memory sort, with both versions using the new sample and distribution code. On the Cray XE6, we used a single Unix process per NUMA node, since its non-uniform memory access makes it inefficient to rely on shared memory between NUMA nodes. On both machines, the gap in distribution time between the pure and mixed versions grows as the number of threads increases, resulting in a speedup of

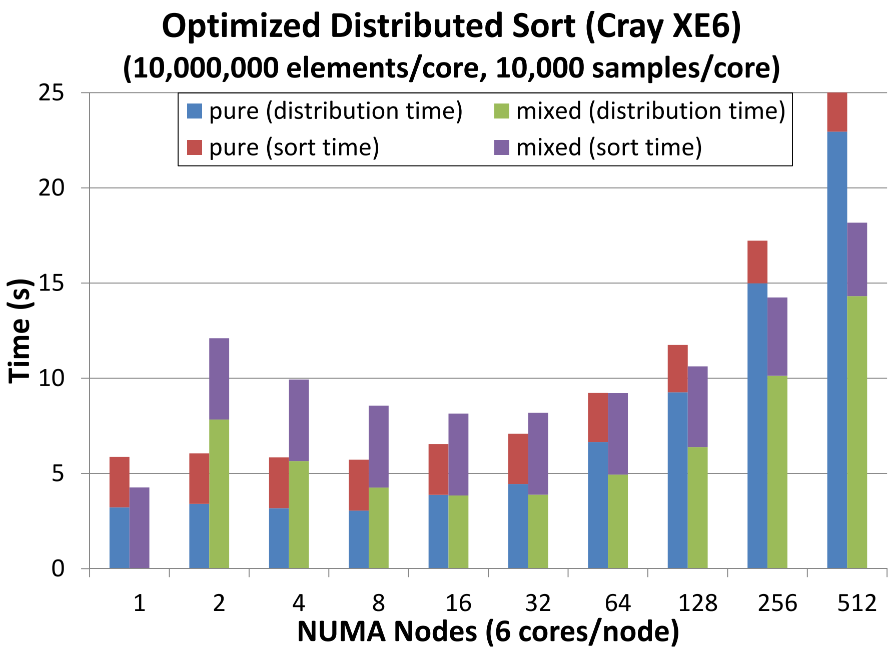


Figure 8.11: Performance of optimized distributed sort on a Cray XE6 with a constant number of elements per thread.

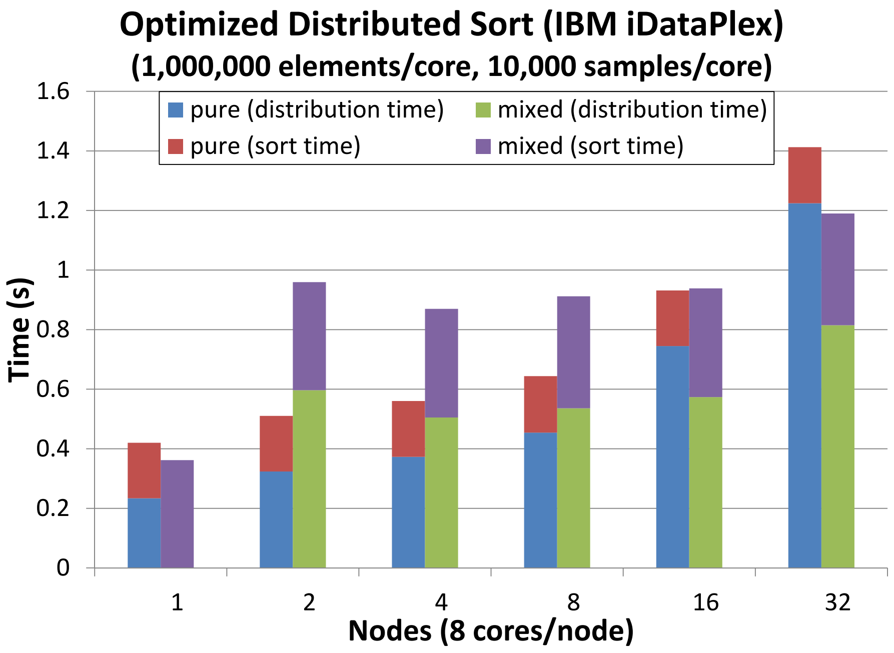


Figure 8.12: Performance of optimized distributed sort on an IBM iDataPlex. We use fewer elements per thread due to memory considerations.

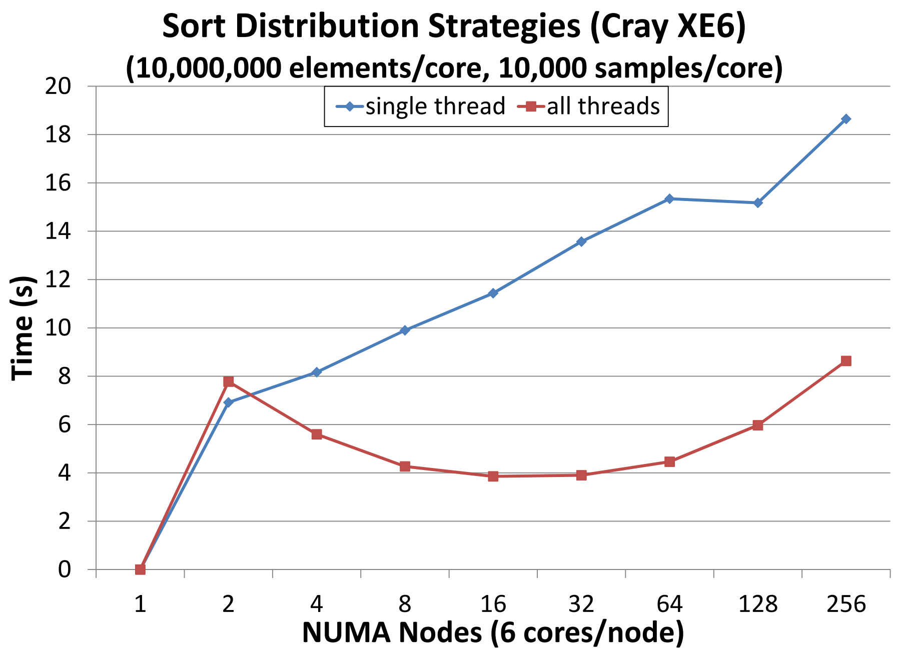


Figure 8.13: Distribution time on a Cray XE6 when using a single thread per team for communication versus all threads in a team.

1.4x for the mixed version over the pure version on both 512 nodes (2048 cores) of the XT4 and 512 NUMA nodes (3072 cores) of the XE6. Figure 8.12 shows performance on the IBM iDataPlex with a smaller problem size, demonstrating similar results as the Cray machines. At 32 nodes (256 cores), the hierarchical version runs about 1.2x as fast as pure sample sort.

As for overall scaling of the algorithm, since the number of elements per node is constant, we expect the sorting phase to remain constant over all numbers of threads, as is the case for both implementations. The distribution phase is dominated by the bucket exchange operation described in §8.1.3.1. While the amount of communication per thread remains constant, the total amount of communication increases linearly and the number of messages increases quadratically, accounting for the increase in distribution time at higher numbers of threads. Sorting in general is not a linear time algorithm, so given that the amount of data per thread is constant, we expect the total time to increase as the number of threads increases.

8.1.3.4 Discussion

The hierarchical, mixed version of sorting that we implemented above aggregates communication between teams, resulting in lower communication costs than the pure sample sort. An important feature of the hierarchical algorithm is that it uses all threads to perform communication in the distribution phase. This is in contrast to a hierarchical sort using the conventional method of mixing MPI with shared-memory parallelism. In the latter, one MPI thread is generally used in

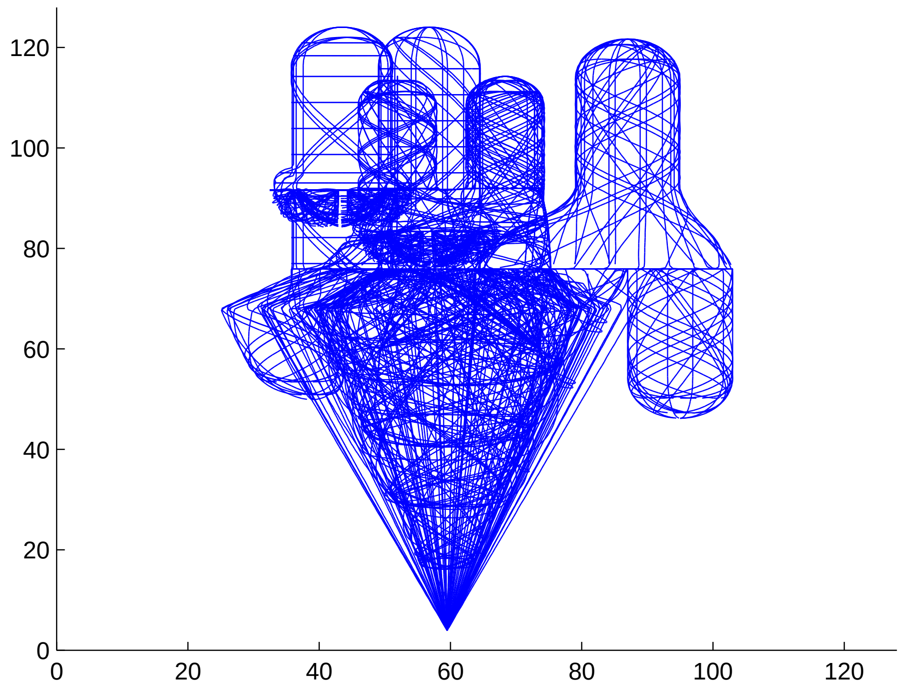


Figure 8.14: Orthographic projection of the heart model. Only a tenth of the fibers are shown for clarity.

Input	Fluid Cells	Particles
Heart	2.10M	606K
256^3 Sphere	16.8 M	1.74M
512^3 Torus	134M	5.53M
512^3 Sphere	134M	7.58M

Table 8.1: Fluid size and number of particles for each particle in cell input.

each shared-memory domain to perform communication between domains, while multiple threads are used for computation. In order to demonstrate the benefit of using multiple threads for communication, we created a new version of our hierarchical sort that uses only a single thread for communication in each shared-memory team. Figure 8.13 compares its distribution time on the Cray XE6 to the version that uses multiple threads, showing that the latter is more than twice as fast as the former at larger thread counts. This demonstrates that a unified hierarchical model of parallelism can provide better performance than one that mixes a distributed model with a shared-memory model.

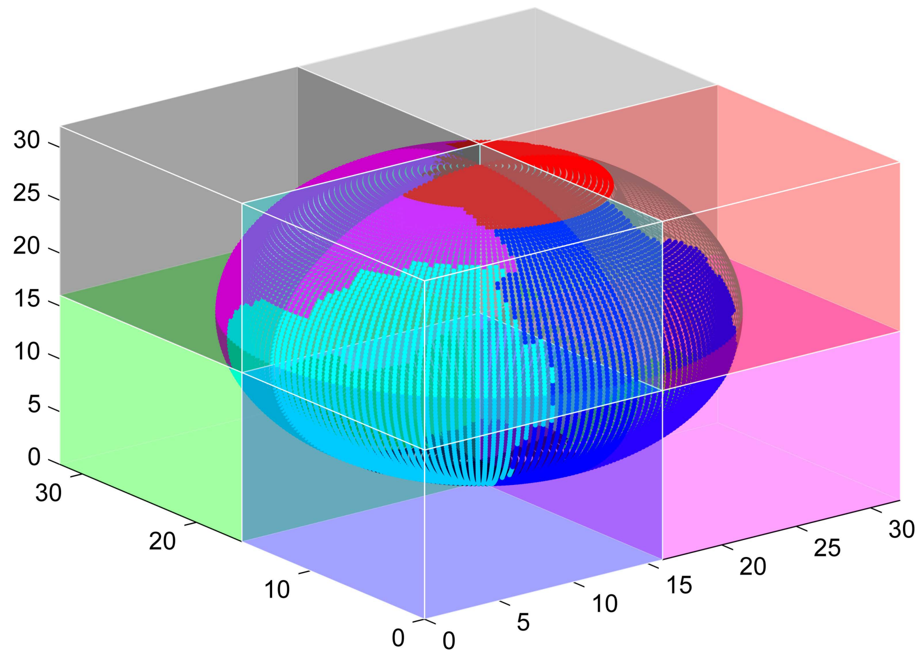


Figure 8.15: Sample input sphere, distributed across eight threads.

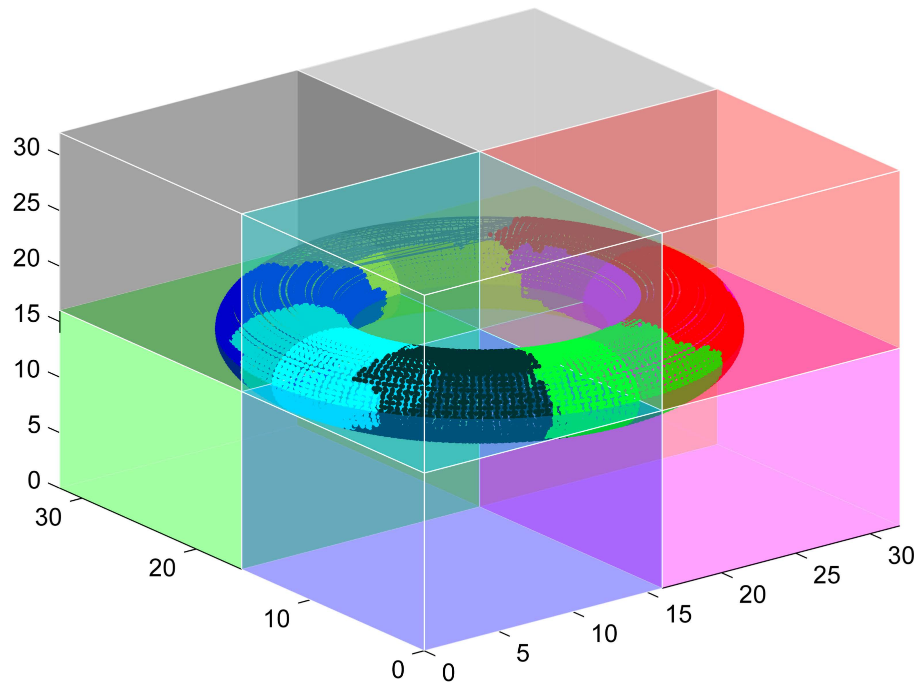


Figure 8.16: Sample input torus, distributed across eight threads.

8.1.4 Particle in Cell

The particle in cell benchmark is inspired by a heart simulation written in Titanium [71, 107, 72]. This simulation uses the immersed boundary method [70] to simulate a heartbeat. The heart is modeled as a set of fibers, consisting of multiple connected fiber points, immersed in a discrete fluid grid, with the fluid cells representing velocity, force, and pressure fields. The simulation is composed of multiple timesteps, each of which consists of the following phases:

1. **Fiber-fiber interaction.** New force values are calculated at each fiber point to represent the contraction of heart muscles.
2. **Fiber-fluid interaction.** Fiber points transfer force to the fluid grid.
3. **Fluid-fluid interaction.** Local forces are used to calculate the velocity of each cell, using the Navier-Stokes equation for incompressible fluids.
4. **Fluid-fiber interaction.** The velocity of each fiber point is calculated from the velocity of the underlying fluid grid, and each point is moved into its new position.

In the particle in cell benchmark, we are concerned with the second phase, where fiber points transfer values to the fluid grid. Thus, the benchmark contains a set of *particles*, corresponding to fiber points, and a set of *cells*, corresponding to the fluid grid. In each iteration, a force value is transferred from each particle to the cell that contains it. Our primary goal is to optimize the communication algorithm in order to reduce running time.

8.1.4.1 Algorithm Overview

Since we are modeling a phase of the heart simulation, we take care to be as true to the heart simulation as possible, starting with the inputs. Figure 8.14 shows the actual heart model used in the simulation. It consists of approximately two million fiber points in a 128^3 fluid grid. Since the model is a fixed size, various synthetic inputs have also been used, such as spheres and tori. Figure 8.15 shows a sample input sphere, and Figure 8.16 shows a sample input torus. The actual inputs we use are a 256^3 sphere, a 512^3 torus, and a 512^3 sphere. Table 8.1 compares their sizes with that of the heart model.

The inputs all share an important characteristic: they are all surfaces embedded in a three-dimensional fluid. As such, there is a lot of space empty of particles, so most fluid cells contain no particles. The third phase of the heart simulation relies on a regularly partitioned fluid grid[1] for load balance, so we cannot simply place particles on the thread that owns their underlying cells. (The particles also move in the heart simulation, but slowly enough so that re-balancing load can be amortized over many iterations. As a result, we ignore particle movement.) The heart simulation uses KMETIS [59] to partition fiber points, with a cost model that takes into account the locality

[1]The current heart implementation uses a one-dimensional fluid partition. However, there is no inherent reason why it cannot use a two- or three- dimensional partition, so we use a three-dimensional partition to allow the code to run on more threads than the number of cells in one dimension.

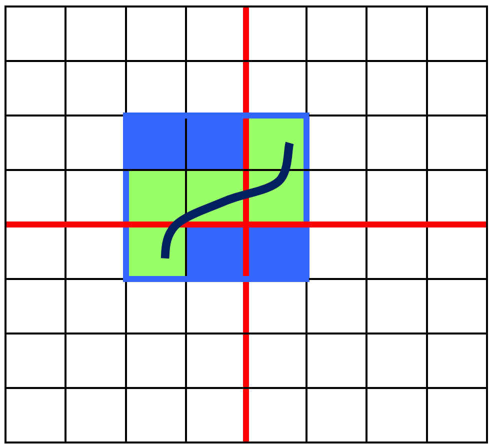


Figure 8.17: Two-dimensional example of a bounding box containing all of a thread's particles.

required in all phases of the simulation. We use the same partitions in our experiments. Figures 8.15 and 8.16 show examples of particle and fluid partitions for eight threads.

Although we attempt to accurately model the heart simulation, we have made a few changes in the interests of simplicity. The most significant change is the number of cells with which each particle interacts. In the actual heart simulation, each fiber point affects force values on each cell in a 4x4x4 grid surrounding the point, for a total of 64 cells. In our benchmark, on the other hand, each particle interacts with only a single cell. This reduces the computational load by a large margin while having only a minor effect on the amount of communication. Since our goal is to optimize communication, the reduction in computation makes this a good benchmark for the communication pattern in the original code, and it does not change the code substantially enough to affect expressiveness.

We also do not include operations that can be amortized over many iterations in the timed portion of the benchmark. These include load balancing, computing target fluid cells, and particle sorting. We do, however, use a two-phase communication algorithm, in which a pointer to target cells is sent to each target processor in every iteration. Particle movement cannot be predicted with certainty, so recomputing target fluid cells cannot be done after a fixed number of iterations. Instead, we send target cell pointers in each iteration, after which the target processor copies the data locally. Other, more complicated algorithms may be possible, but we do not explore them here.

Multiple particles can be located in each fluid cell, and these particles can be located on different processors. As such, updates to each fluid cell must be synchronized. We use the update-by-owner model [94], in which the owner of each fluid cell processes updates to that cell. On each thread, particles update local copies of their target cells. Each target cell is then sent to the thread that owns it, which updates the canonical version of that cell. This scheme involves reordering particle updates; updates are associative and commutative, modulo floating-point precision, so reordering is valid.

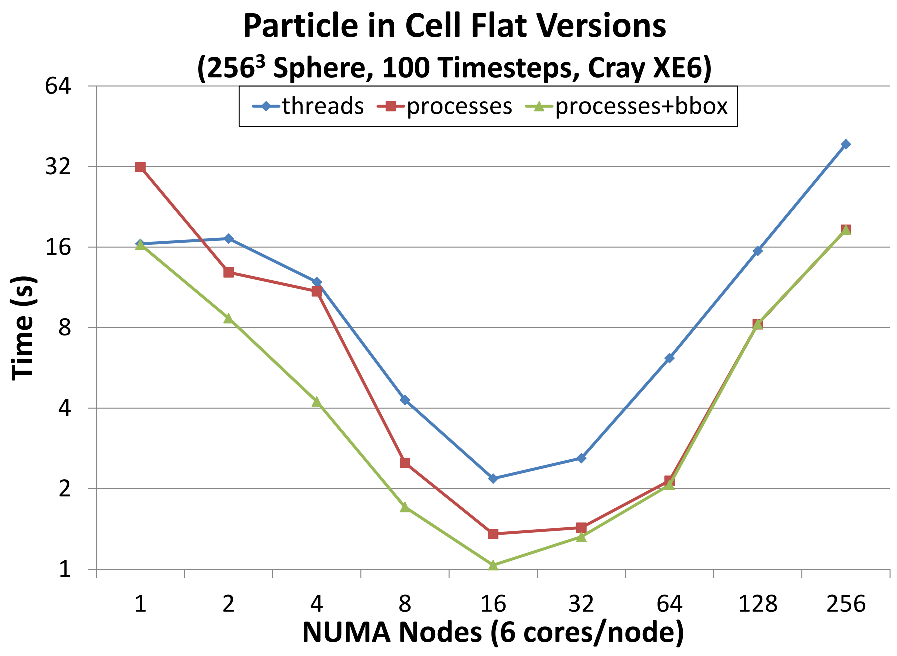


Figure 8.18: Performance of flat particle in cell using shared-memory threads on a NUMA node versus processes that do not share memory.

The main data structures in the benchmark include an array of points on each thread, two copies of the fluid cells that the thread owns, and two copies of the target fluid cells underlying the thread's points. The latter consist of a cubic three-dimensional bounding box, similar to the square two-dimensional box pictured in Figure 8.17. The intersection of this bounding box with the portion of the fluid grid owned by each thread is communicated in each iteration of the algorithm. Titanium's powerful array library allows these intersections to be computed trivially and provides a simple mechanism for constructing a view of a subset of an array. Using these array views, the bounding box data structure itself is restricted to produce the required data structures for communication, resulting in no extra memory usage aside from array descriptors.

The timed portion of the resulting algorithm consists of multiple iterations of the following steps on each thread.

1. Clear target fluid cells (i.e. the bounding box).

2. Process particles, updating target fluid cells.

3. Send pointers to target-fluid subarrays to the other threads.

4. Barrier synchronization to ensure previous steps are completed on all threads.

5. Copy incoming fluid subarrays locally and process updates to fluid owned by this thread. Non-blocking communication is used to overlap communication with computation and other communication.

6. Swap duplicated data structures.

We refer to our implementation of this algorithm as the *flat* version, since it contains no shared-memory optimizations.

Figure 8.18 shows the execution time of the flat algorithm on a Cray XE6 with a 256^3 sphere as input. The code was run with each thread in its own Unix process, so that no threads share memory, labeled as *threads* in the figure. We also ran with one Unix process per NUMA node, labeled as *processes*, so that groups of six threads share memory. The latter performs significantly worse than the former, twice as slow on average, since our runtime layer is not optimized for shared memory. This illustrates the gap that any shared-memory optimizations must overcome in order to be effective.

The third version in Figure 8.18, *processes+bbox*, is also run with one thread per process, but it includes a bounding box optimization. Instead of computing a single cubic bounding box for all points owned by a thread, we compute separate cubic bounding boxes for each target thread, represented in two dimensions by the light green area in Figure 8.17. This decreases communication volume, resulting in significant performance benefits. All three versions cease scaling beyond 16 NUMA nodes.

8.1.4.2 Hierarchical Optimizations

Starting from the flat algorithm without the bounding box optimization, we wrote hierarchical versions to take advantage of shared memory. Each set of threads that share memory is placed into its own teams, and all of our hierarchical versions partition the particles and fluid grid by the number of teams, rather than the total number of processors.

Our initial version, referred to as *replicated* in Figure 8.19, replicates a team's portion of its fluid grid as well as its target fluid cells on all threads in the team. It also distributes the team's particles randomly to its threads. In step 2 of the algorithm, the threads update their own target fluid copies with their particles, after which a reduction is performed to combine their target fluid grids. A single thread from the team then sends pointers to the combined target grids to the other teams in step 3. After the barrier synchronization, that same thread launches non-blocking copies in step 5. These copies are then split among all threads in the team, which wait for them to complete and update their own fluid cells. The fluid grids are then combined using a final reduction.

A second version, which we refer to as *split*, evenly divides the fluid grid across a team's threads. Target fluid cells are still replicated. Now in step 5, all threads process all incoming target fluid cells, updating only the portion of the fluid grid that they own. This eliminates the reduction in that step, somewhat improving performance.

In order to eliminate the reduction in step 2, we needed to divide a team's target-fluid grid among its threads. This in turn requires avoiding overlap between the underlying fluid cells of each thread's particles. We thus sort a team's particles by position before dividing them evenly

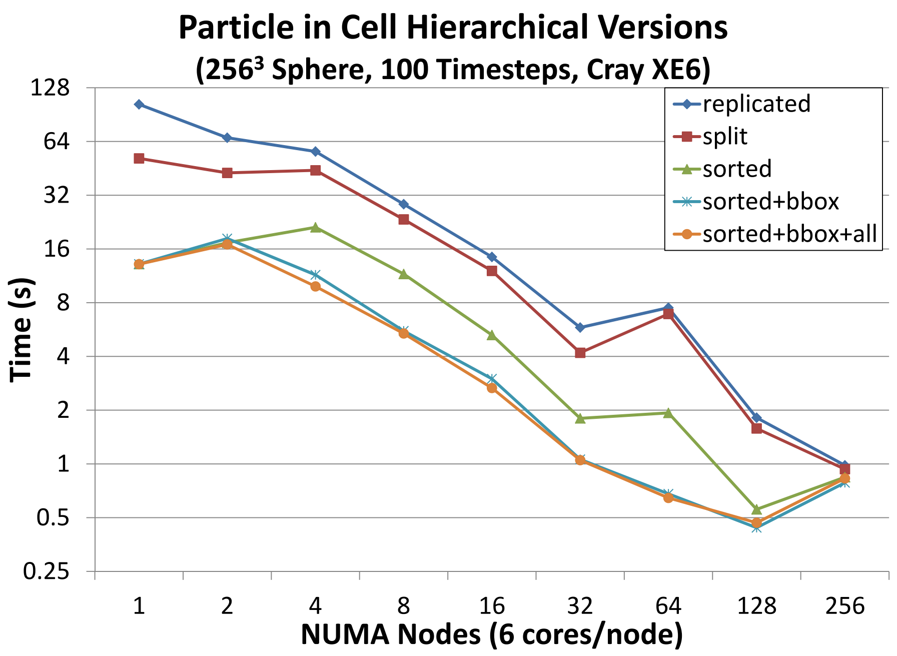


Figure 8.19: Performance of various hierarchical particle in cell algorithms on a Cray XE6.

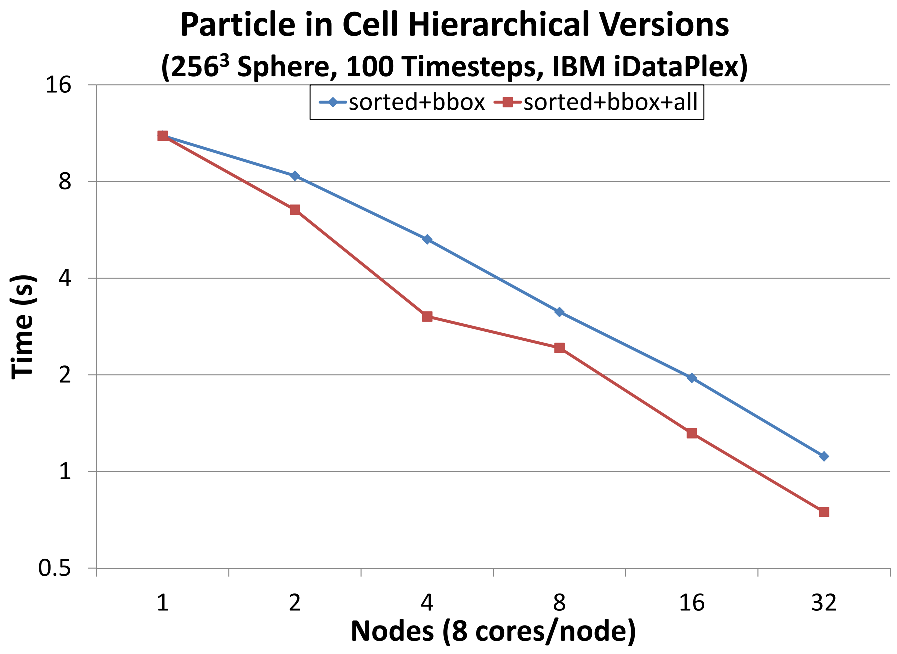


Figure 8.20: Performance of various hierarchical particle in cell algorithms on an IBM iDataPlex.

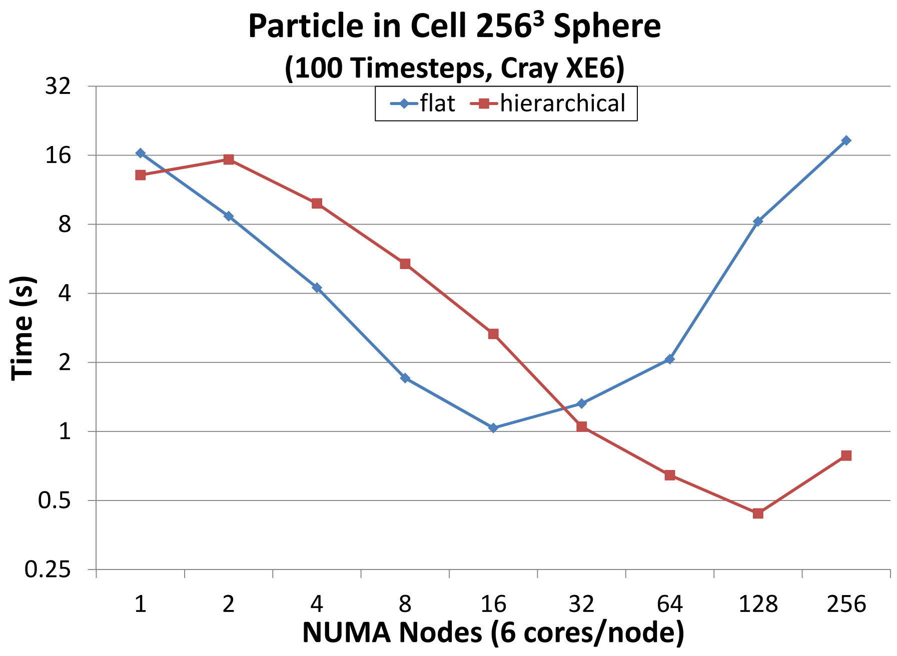


Figure 8.21: Performance of a 256^3 sphere on a Cray XE6 using the best hierarchical particle in cell algorithm versus the best flat algorithm.

among the threads. Now at most one target fluid cell on each thread can overlap another thread's target grid. We assign such a cell to the lowest ID thread that has particles that update that cell and use update-by-owner to combine updates to the cell. We call the resulting algorithm *sorted*, and it significantly improves performance over the split version. We have found sorting overhead to be very low, averaging around 5% of execution time when sorting every 100 timesteps. This is with an unoptimized, sequential sort, and we believe that the overhead can be further reduced using a tuned parallel sort.

Our next version, *sorted+bbox*, applies the bounding box optimization described in the flat case. This also results in a significant performance benefit.

The final optimization we applied was to use all of a team's threads to send pointers in step 3 and to launch non-blocking copies in step 5. The latter can involve packing operations, so parallelizing them can improve performance. Figure 8.19 shows that the resulting code, labeled *sorted+bbox+all*, provides similar performance as the previous version on the Cray XE6. On the other hand, it provides significantly better performance on the IBM iDataPlex, as shown in Figure 8.20.

Figure 8.21 compares performance of the best flat algorithm to the best hierarchical algorithm on the Cray XE6, with a 256^3 sphere as input, and Figure 8.22 shows performance on the IBM iDataPlex. The flat algorithm does not scale beyond 16 nodes on the Cray machine and 8 nodes on the IBM machine, while the hierarchical algorithm scales up to 128 and 32 nodes, respectively. On the other hand, the flat algorithm performs about twice as fast as the hierarchical version up to

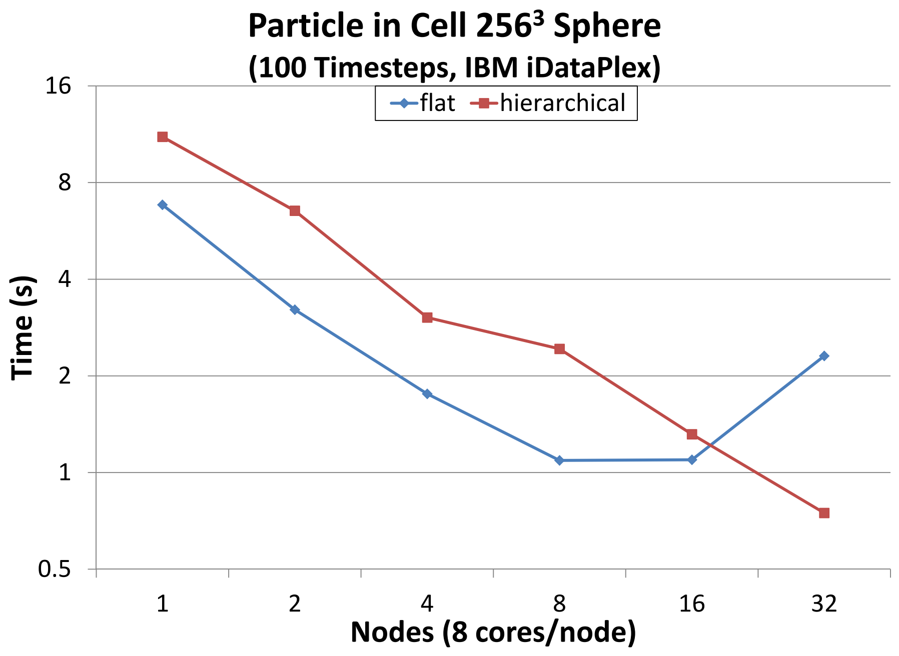


Figure 8.22: Performance of a 256^3 sphere on an IBM iDataPlex using the best hierarchical particle in cell algorithm versus the best flat algorithm.

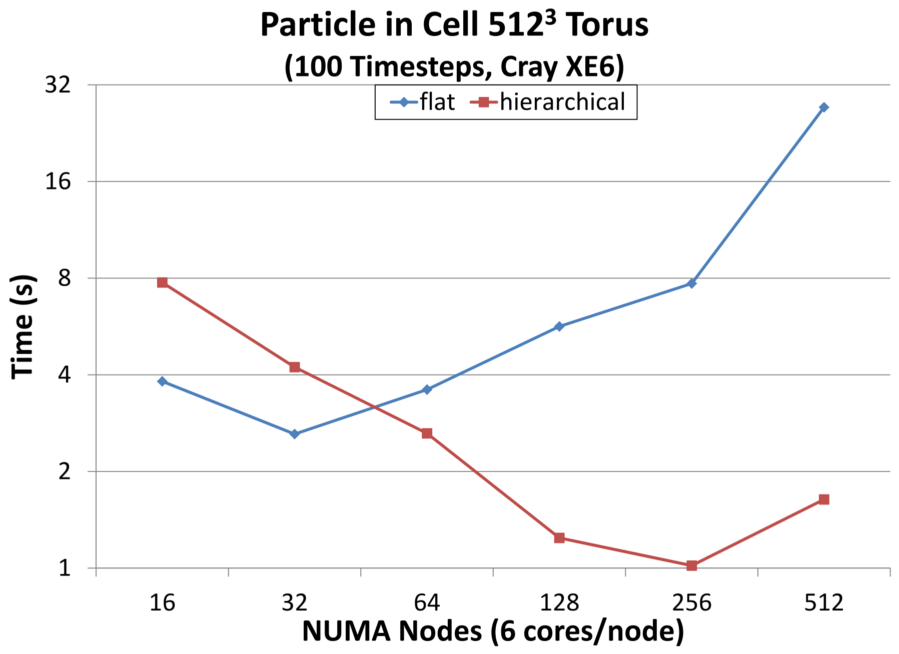


Figure 8.23: Performance of a 512^3 torus on a Cray XE6 using the best hierarchical particle in cell algorithm versus the best flat algorithm.

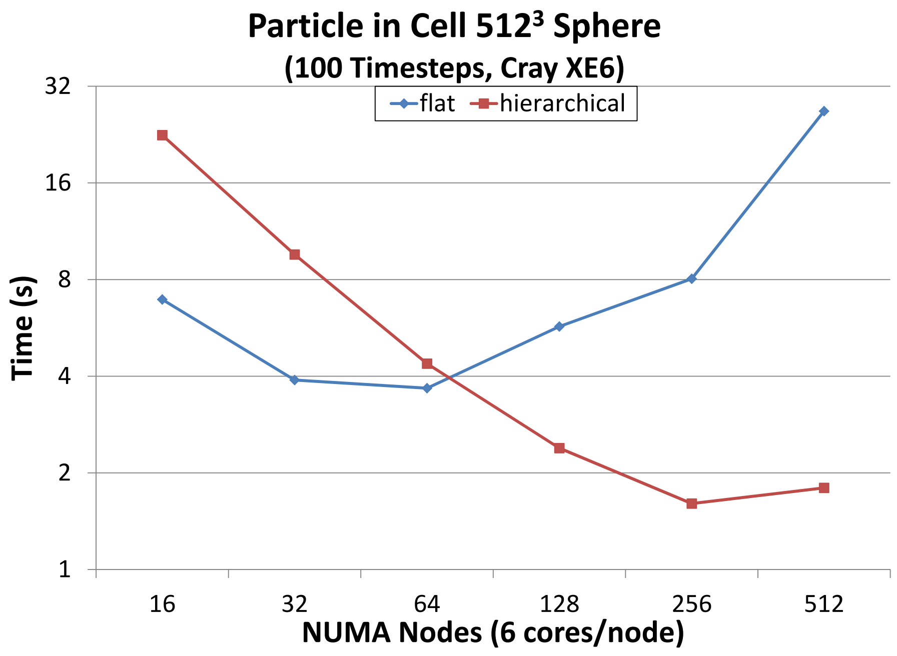


Figure 8.24: Performance of a 512^3 sphere on a Cray XE6 using the best hierarchical particle in cell algorithm versus the best flat algorithm.

the former's scaling limits. The hierarchical algorithm requires four times as many processors to improve running time beyond the best performance of the flat algorithm.

Figures 8.23 and 8.24 show performance on the Cray machine with the 512^3 torus and 512^3 sphere[2]. They demonstrate similar results as the 256^3 sphere.

8.1.4.3 Discussion

The particle in cell benchmark proved to be a less than ideal candidate for hierarchical optimizations. Below the scaling limit for the flat version, the hierarchical optimizations were unable to overcome the performance gap of shared memory shown in Figure 8.18. The primary reason for this is that the benchmark overlaps communication with communication and computation, reducing the actual cost of communication to overall running time. Above the flat scaling limit, both communication volume and computation time decrease to the point where overlapping is no longer sufficient. It is only here that hierarchical optimizations improve performance, although at a significant programming cost.

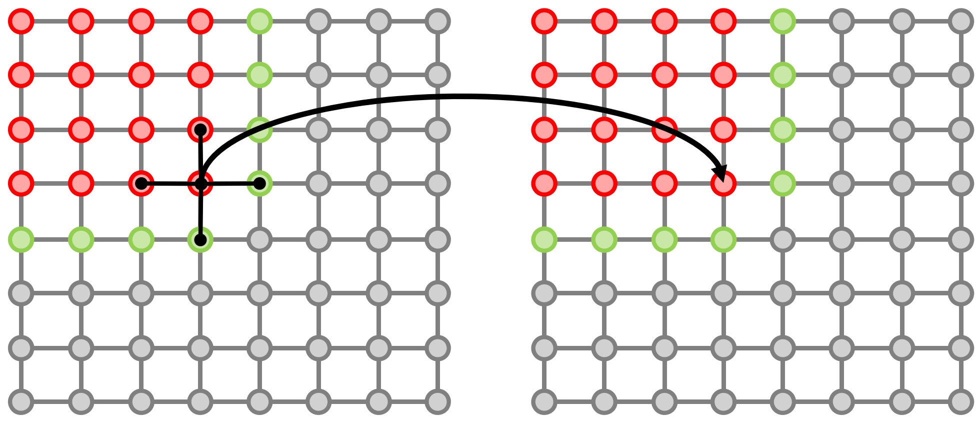


Figure 8.25: A five-point stencil in two dimensions. The ghost zones of the node that owns the top left portion of the grid are in light green.

8.1.5 Stencil

A *stencil* is a nearest-neighbor computation over a structured n-dimensional grid. Stencils are generally used to numerically solve partial differential equations and consist of multiple iterations in which the value of each grid point is updated as a function of its previous value and those of its neighboring points. Figure 8.25 illustrates a five-point stencil on a two-dimensional grid.

In this benchmark, we examine a seven-point stencil in three dimensions, the three-dimensional analogue of the five-point stencil pictured in Figure 8.25. The stencil we use is an abstraction of that used to solve the *heat equation*,

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right),$$

with fixed boundary conditions at the edges of the grid. We use the *Jacobi* iteration strategy, in which there are two copies of the grid. In each iteration, one grid is read from and the other is written to. As such, this is an *out-of-place* computation.

A wide variety of work has been done to optimize stencil computations on shared-memory multiprocessors, including communication-avoiding optimizations [69, 105], auto-tuners [27, 56], standalone domain-specific compilers [97], and embedded domain-specific compilers [20, 55, 58]. Rather than reproduce this body of work, we wish to take advantage of existing libraries in order to perform the shared-memory portion of the stencil computation.

We thus partition the grid among shared-memory nodes or NUMA nodes. As shown in Figure 8.25, points at the edge of a node's partition depend on points on other nodes; we refer to the set of required points on each neighboring node as a *ghost zone*. Each node has at most six ghost zones, one in each direction in three-dimensional space, and ghost zones must be communicated between nodes after each iteration. In a seven point stencil, all ghost zones can be transferred in parallel, so we overlap ghost-zone communication on each node.

[2]Due to memory constraints, we were unable to run these problem sizes on the IBM machine.

In our stencil implementation, we use Titanium to perform all inter-node communication but use an external library to perform the on-node, shared-memory computation. Since those libraries have their own tuning frameworks, we are interested only in optimizing performance of the Titanium benchmark, independent of the performance of the external libraries. As such, we use naïve, untuned versions of the external code for simplicity.

For each external library, we compare three versions of the Titanium code:

- **seq+*lib***: A single Titanium thread is used on each node, while multiple threads are used in the external library. This is similar to the common case of mixed MPI and shared-memory parallelism, where a single MPI process is used on each node.
- **par+*lib***: Multiple Titanium threads are used on each node, but only one is used for communication, while the external library uses multiple threads. This strategy may be used as part of a larger distributed application, as it is simpler than using multiple threads for communication. It also demonstrates any overhead from using multiple Titanium threads on a node.
- **par+*lib*+all**: Multiple Titanium threads are used on each node for communication, and the external library also uses multiple threads. This is a true hierarchical version of the benchmark, as it takes advantage of all available parallelism.

We use both an OpenMP [80] stencil library and one that uses POSIX threads (Pthreads).

8.1.5.1 OpenMP

The OpenMP library we use is an OpenMP version of the Stencil Probe microbenchmark [93, 57, 30]. Since on-node parallelism is handled by the OpenMP compiler and runtime, we call the library from a single Titanium thread on each node, even if there are multiple Titanium threads on each node. The par+omp and par+omp+all versions of the code therefore use two team hierarchies, one that divides the threads into shared-memory teams, and another that divides them into teams with one thread from each node, as in §8.1.3.3:

```
Team team = Ti.defaultTeam();
team.initialize(false);
Team oTeam = team.makeTransposeTeam();
oTeam.initialize(false);
```

The structure of the stencil computation is then as follows:

```
for (int i = 0; i < numIter; i++) {
  // copy ghost zones
  ...
  // call external library
  partition(oTeam) {
    { stencilOMP(gridA, gridB, dimx, dimy, dimz); }
  }
```

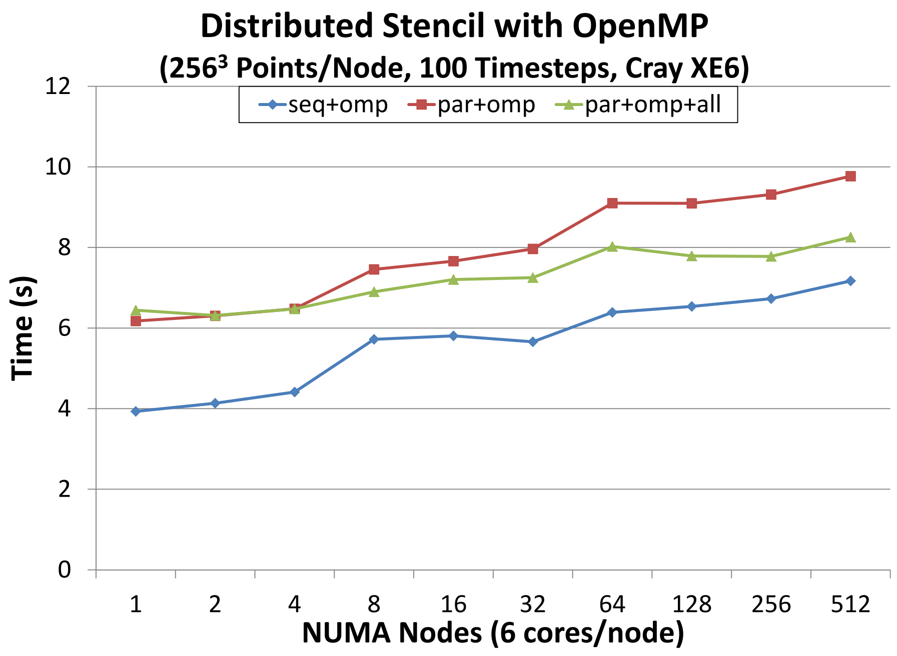


Figure 8.26: Weak scaling performance of stencil on a Cray XE6 using OpenMP for shared-memory parallelism.

```
 8    // wait for local computation to finish
 9    teamsplit(smpTeam) {
10      Ti.barrier();
11    }
12    // swap pointers
13    ...
14  }
```

In each iteration, ghost zones are first copied. Then one thread from each shared-memory node calls the external OpenMP library, while the other threads wait on a node-local barrier. Finally, the data structures are swapped for use in the next iteration.

As the above code demonstrates, team constructs facilitate composition with external libraries. However, the interaction between Titanium's shared-memory threading, which is built on Pthreads, and OpenMP results in a significant performance degradation. Figure 8.26 shows weak scaling (constant problem size per thread) performance of the three code variants on a Cray XE6, with 256^3 points per NUMA node. The gap between seq+omp and par+omp demonstrates the performance hit from mixing Pthreads and OpenMP. We tested two implementations of the local barrier in line 10 above, one based on mutexes and another that uses Pthread barriers. Both resulted in the same performance on the Cray XE6.

Figure 8.27 shows performance of the stencil variants on an IBM iDataPlex, demonstrating an even larger gap between code that mixes Pthreads with OpenMP and code that does not. The results

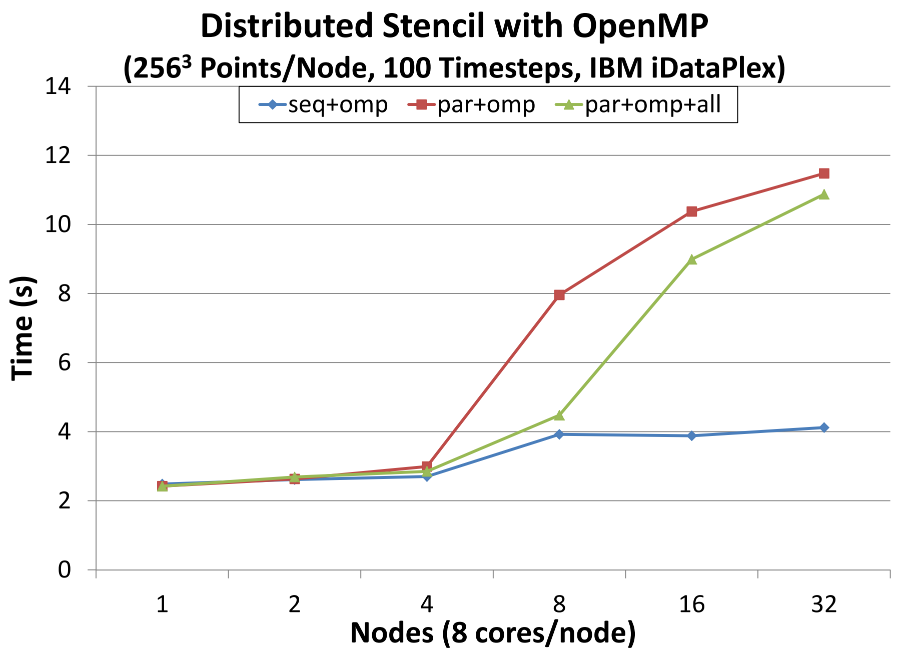


Figure 8.27: Weak scaling performance of stencil on an IBM iDataPlex using OpenMP for shared-memory parallelism.

shown are for a barrier implementation that uses Pthread barriers; the mutex implementation was an order of magnitude worse on this platform.

While the interaction between Pthreads and OpenMP results in poor overall performance, the results do demonstrate that a hierarchical stencil code, one that uses multiple threads for communication on each node, results in better performance than a flat version that uses only a single thread per node. We believe that a port of Titanium to the Lithe runtime [83, 82, 81] would eliminate the performance penalty from mixing Pthreads and OpenMP, resulting in the hierarchical par+omp+all version running faster than the seq+omp variant.

8.1.5.2 POSIX Threads

Since Titanium uses POSIX threads, or Pthreads, for shared-memory parallelism, the combination of Titanium with a Pthreads stencil library should not result in the slowdowns experienced in the OpenMP case. We use code generated from an auto-tuner written by Kaushik Datta [27]. Unlike in the OpenMP implementation, the par+pthread and par+pthread+all variants here share threads between Titanium and the external library. Thus, all threads call into the stencil library, passing their thread ids and the number of threads in the team as arguments. The code then is as follows:

```
for (int i = 0; i < numIter; i++) {
  // copy ghost zones
  ...
```

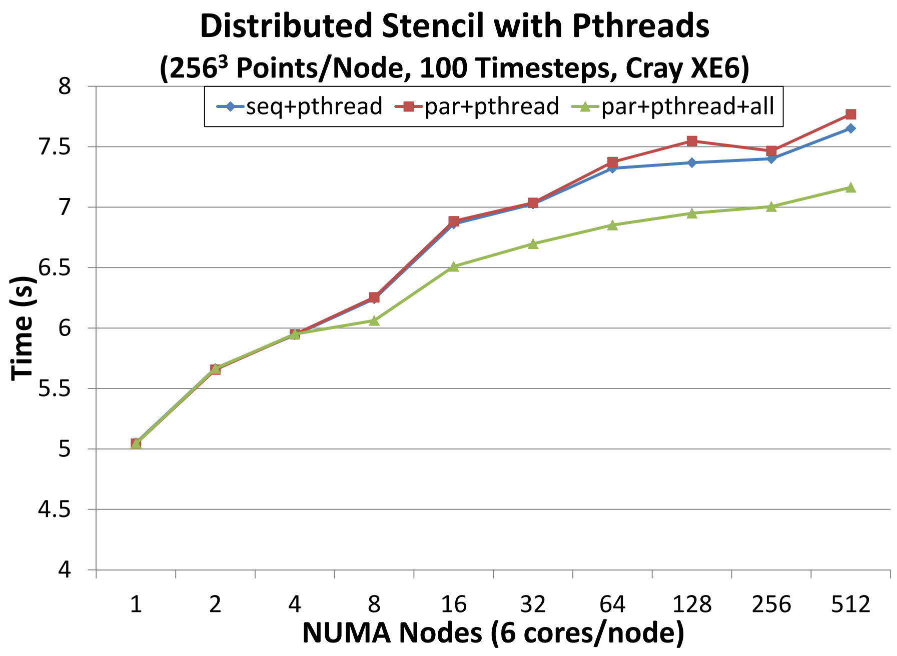


Figure 8.28: Weak scaling performance of stencil on a Cray XE6 using Pthreads for shared-memory parallelism.

```
  teamsplit(smpTeam) {
    // call external library
    stencilPthread(gridA, gridB, dimx, dimy, dimz,
                   Ti.thisProc(), Ti.numProcs());
    // wait for local computation to finish
    Ti.barrier();
  }
  // swap pointers
  ...
}
```

Team constructs help mainly in keeping track of which threads share memory, as well as providing local collective operations such as barriers.

Figures 8.28 and 8.29 show weak scaling performance of the stencil variants on a Cray XE6 and an IBM iDataPlex, respectively. The gap between the seq+pthread and the other versions at lower node counts on the IBM iDataPlex is due to overhead in the Titanium runtime system, as described in §8.1.4. On both machines, the hierarchical par+pthread+all version outperforms the other two variants at higher node counts, demonstrating the performance benefits of hierarchical code. It improves performance over seq+pthread by up to 7% on the Cray machine and 14% on the iDataPlex.

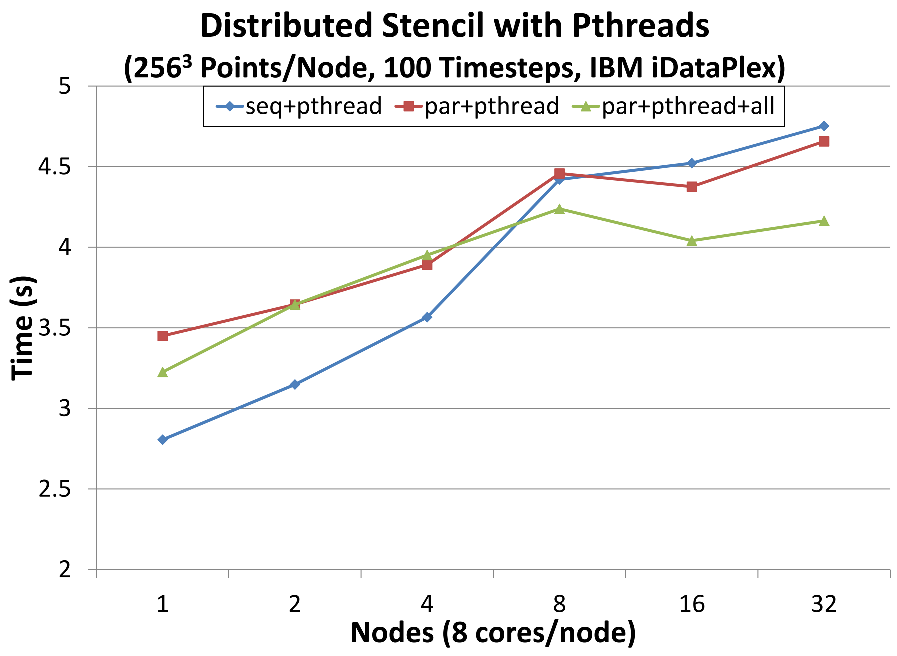


Figure 8.29: Weak scaling performance of stencil on an IBM iDataPlex using Pthreads for shared-memory parallelism.

In both the Pthread implementation here and the OpenMP implementation above, the hierarchical par+*lib*+all variants only perform better than the par+*lib* variants on node counts greater than four. This is due to the fact that at lower node counts, each node only has at most two neighboring nodes, providing less opportunity for parallelizing communication. Also, we overlap communication with communication in all variants, so the main benefit from using multiple threads for communication is to parallelize the packing of data required before each transfer. Despite this limited opportunity for parallelization, the hierarchical versions still perform better than the flat par+*lib* variants.

8.2 Dynamic Alignment of Collectives

We have verified on many different test cases that the dynamic enforcement system does detect alignment errors in practice without requiring any programmer annotations. Consider the following program, reproduced from §4.2.2.

```
if (Ti.thisProc() == 0) {
  fakeBarrier();
} else {
  fakeBarrier();
}
```

```
Ti.barrier();
```

Upon running this code, the following error is produced under strict alignment in debugging mode in addition to the usual Java-like stack trace:

```
ti.lang.Alignment.AlignmentError: collective alignment
 failed on processor 1 at foo.java:10:8
last location: else branch at foo.java:5:12
last location on processor 0: then branch at foo.java:5:12
previous location: none
```

The error message directs the user to the exact location that caused alignment to fail, instead of just providing the location of the misaligned collective itself.

Since Titanium currently enforces alignment statically, no Titanium application has alignment errors, and we could not test its effectiveness on real-world programs. However, we were able to determine the performance cost of dynamic checking on two platforms: an eight-core (two-processor, four-core) Intel Xeon E5435 shared-memory multiprocessor (SMP) and *Jacquard*, a cluster at NERSC that consisted of dual-processor 2.2GHz Opteron nodes with an InfiniBand interconnect.

Five program versions were compared:

- **static**: no dynamic checking
- **strict**: strict alignment scheme, default mode with no execution history list
- **strict/debug**: strict alignment scheme, debugging mode with execution history list
- **weak**: weak alignment scheme, default mode with no execution history list
- **weak/debug**: weak alignment scheme, debugging mode with execution history list

8.2.1 Collective Performance

We first tested the performance of three of the primitive collectives by repeatedly invoking them inside a loop. For the dynamic alignment schemes, a single loop iteration includes an update to the execution hash (and list for the debugging modes), and for the broadcast test, an additional execution hash/list update for the source thread of the broadcast. Each loop iteration also includes the hash comparison code associated with a collective operation, which consists of a broadcast, comparison, and conditional.

Figure 8.30 shows the relative loop iteration time for the broadcast, barrier, and exchange tests on the SMP machine for up to 8 processors. The broadcast is of a single 32-bit integer, and the exchange is of 32-bit integers among all threads. On average, barriers in the dynamic schemes take about 2.7 times as long as the static version, broadcasts take 2.5 times as long, and exchanges 70% longer. The dynamic debugging versions consistently were slower than the default dynamic versions by an average of about 20%.

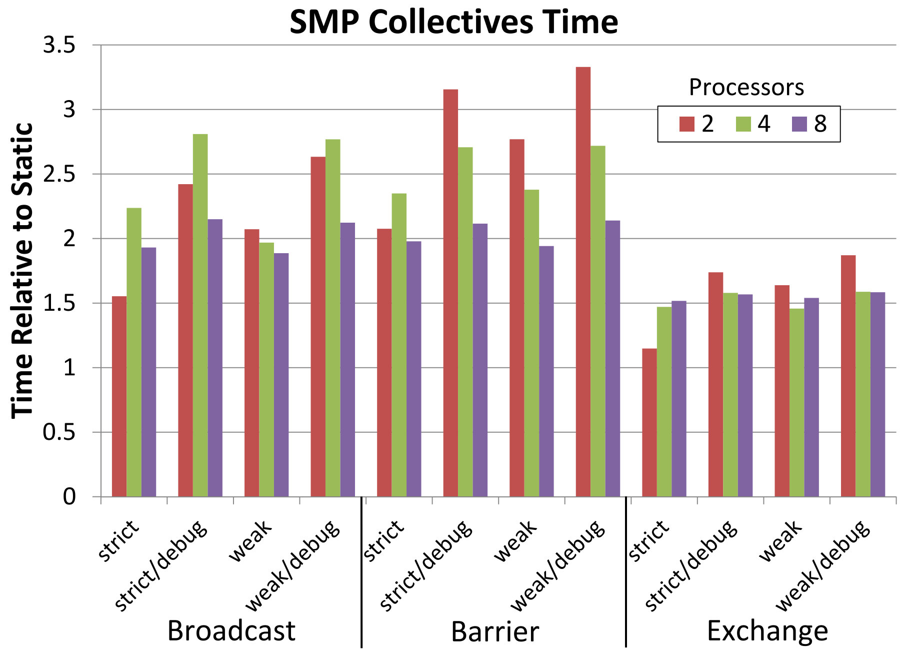


Figure 8.30: Relative collective performance on the SMP machine.

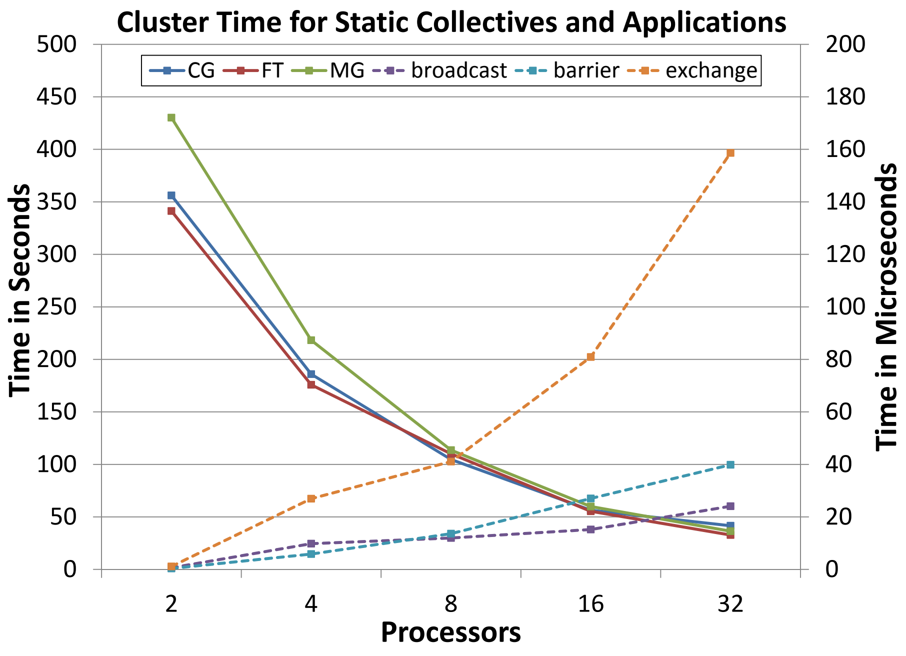


Figure 8.31: Collective and application performance on the cluster machine in the static case. Application times are plotted with respect to the left vertical axis and collective times with respect to the right.

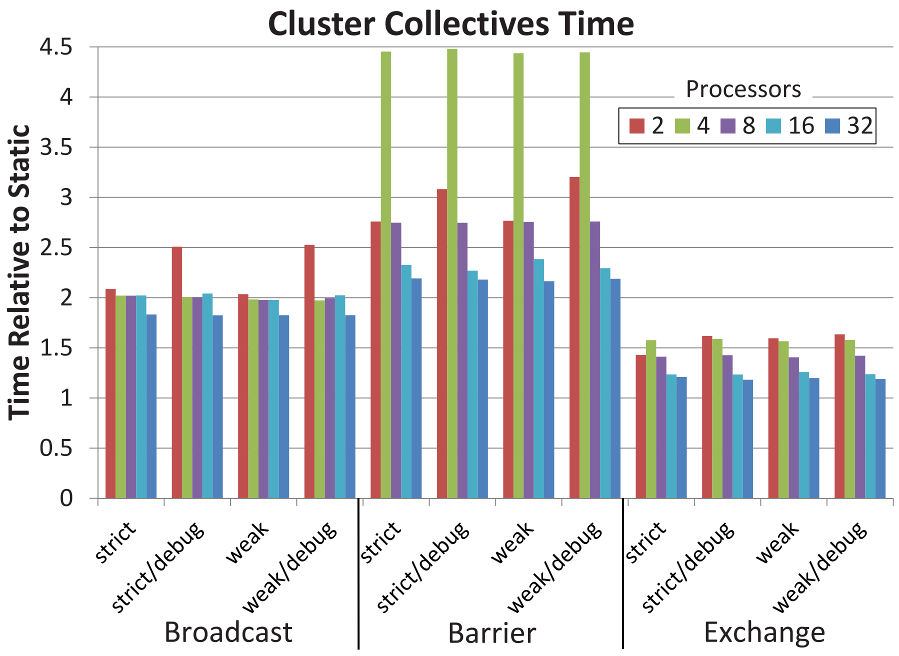


Figure 8.32: Relative collective performance on the cluster machine.

Figure 8.31 shows the loop iteration times for each collective in the static case on the cluster machine for up to 32 processors, and Figure 8.32 shows the dynamic times relative to the static time. The broadcast and exchange are as in the SMP case. The jump in barrier time at four processors is due to it being the fewest number of processors to require inter-node communication. On average, barriers in the dynamic schemes take about three times as long as the static version, broadcasts take twice as long, and exchanges 40% longer. There is little discernible difference between the various dynamic versions.

On both machines, the overhead of dynamic checking decreases for each collective operation as the number of processors increases. In particular, the cost of an unchecked broadcast trends to about half that of a barrier on the cluster machine and becomes negligible compared to the cost of an exchange. Since checked operations include an extra broadcast, we expect that for a large number of processors, 32-bit broadcasts would take twice as long with dynamic checking compared to static, barriers would take 1.5 times as long, and 32-bit exchanges would take about the same amount of time. Since the overhead of checking is constant for a given number of processors, we expect larger collective operations, such as multiword or vector broadcasts, to demonstrate even smaller slowdowns.

8.2.2 Application Performance

We also tested three of the NAS Parallel Benchmarks [9] in Titanium [29, 28]: conjugate gradient (CG), Fourier transform (FT), and multigrid (MG). (We use a version of CG without teams here

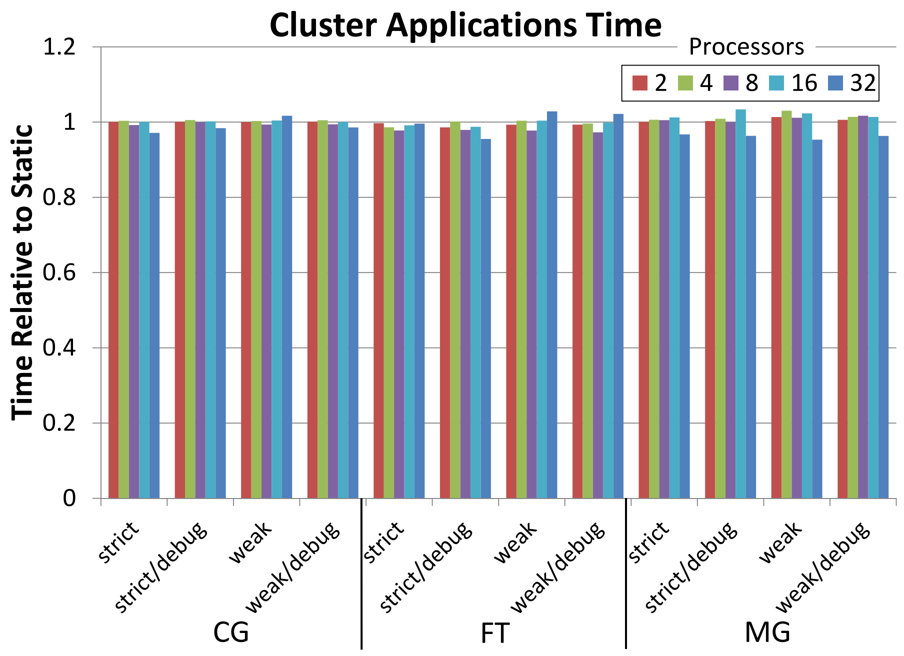


Figure 8.33: Relative application performance on the cluster machine.

and in the following sections. See §8.4.1.1 for more details.) Due to memory constraints, the three benchmarks used class B, A, and B sized problems, respectively, though the parameters were tweaked to increase running time.

In general, parallel applications do not perform collectives in their inner, compute-intensive loops, nor do such loops tend to have global effects. Thus, even though collectives are slower under dynamic alignment and tracking operations have some additional cost, we expect them to be executed rarely and do not expect them to drastically affect application performance. Figure 8.31 shows the running time for the NAS Parallel Benchmarks on the cluster machine in the static case for up to 32 processors. Figure 8.33 shows the relative dynamic times, demonstrating that for these applications, the dynamic checks have no effect on performance. The same results also occurred on the SMP, though the graphs have been omitted for brevity.

8.2.2.1 Analysis

In order to understand why dynamic checking does not appear to significantly affect application performance, we ran experiments to determine how much impact it should have. In Table 8.2, we report the amount of time it takes for each alignment operation on the two machines, including the time to update the alignment hash, the time to update the history and hash in debugging mode, the time to perform a history save and restore pair for weak alignment, and the time to perform a hash check. The latter varies depending on the number of processors, so we report the maximum time on all processor counts we tested.

Operation	SMP Time	Cluster Time
Update	4.0 ns	12.8 ns
Debug Update	31.4 ns	152.3 ns
Save/Restore	10.3 ns	87.2 ns
Max Check	1.6 μs	47.5 μs

Table 8.2: Time for alignment update and check operations on each machine.

	Operation Count			Max Total Time	
	Updates	Saves/Restores	Checks	SMP	Cluster
CG	1844	924	2729	4.4 ms	0.13 s
FT	1835	1216	1218	2.0 ms	0.05 s
MG	100530	69248	28320	48.9 ms	1.37 s

Table 8.3: Number of operations and calculated overhead for each benchmark.

Table 8.3 shows the number of times each operation occurs in each benchmark. Using these numbers and the times from Table 8.2, we conservatively estimate the maximum effect on running time for each machine. The worst case is 1.37 seconds for the multigrid application on the cluster, which is less than 5% of total running time on 32 processors. The results in Figure 8.33 show that the actual effect is even less than this.

In general, most applications try to avoid collectives since they limit scalability. As a result, they would not be affected much by the overhead of dynamic checking at each collective. Applications that spend a significant fraction of their time in small collectives can expect this portion of their execution time to as much as double for large numbers of processors, as we argued in §8.2.1. If this cost is too high, users can turn of checking as noted in §4.2.3. In addition, the optimizations described in §4.2.2.1 should significantly reduce the performance impact of dynamic checking.

We also determined how much space the execution history list uses in debugging mode. Since entries are cleared from the list at each collective, the number of entries in the list when performing a collective is equal to the number of control-flow decisions that affect execution of the collective since the previous collective executed. We expect this number to be relatively small and the execution history to use little space as a result. Our tests showed that the space used on each thread was 0.8 KB. 0.3 KB, and 90 KB for CG, FT, and MG, respectively, matching our expectations.

8.3 Pointer Analysis

The pointer information computed in §6 can be applied to multiple analyses and optimizations for parallel programs. In this section, we take a look at two clients, locality inference and sharing inference, and the effect of pointer analysis on them. Both applications are compared to existing constraint-based inferences on the set of benchmarks below. The constraint-based implementations do not distinguish between allocation sites, so the pointer-based implementations should perform

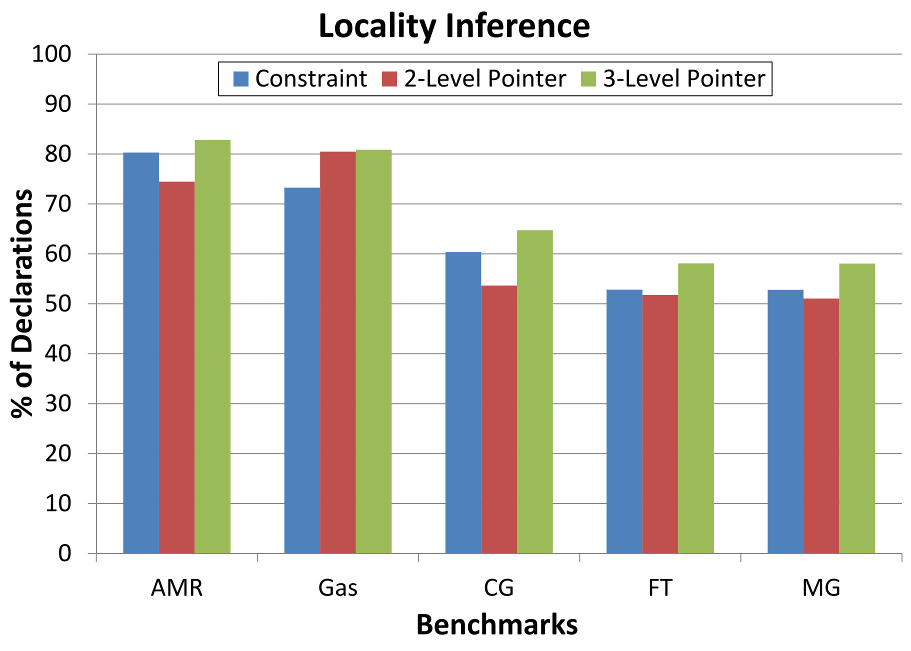


Figure 8.34: Locality inference results for some Titanium programs.

better at maximum precision. This advantage would disappear were the constraint-based implementations modified to separate allocation sites.

We use the following benchmarks to evaluate our analysis:

- **AMR** [103] (7581 lines): Titanium implementation of the Chombo adaptive mesh refinement suite [7].
- **Gas** [10] (8841 lines): Hyperbolic solver for a gas-dynamics problem in computational fluid dynamics using adaptive mesh refinement, by Peter McQuorquodale and Phillip Collela.
- **CG** (1595 lines): NAS conjugate gradient benchmark in Titanium.
- **FT** (1192 lines): NAS Fourier transform benchmark in Titanium.
- **MG** (1952 lines): NAS multigrid benchmark in Titanium.

The line counts for the above benchmarks underestimate the amount of code actually analyzed, since all reachable code in the 37,000 line Titanium and Java 1.0 libraries is also processed.

8.3.1 Locality Inference

Pointer information can be used to infer the minimal width of a particular reference or expression. In particular, if an expression e of reference type evaluates to the abstract set S, then its minimal width is:

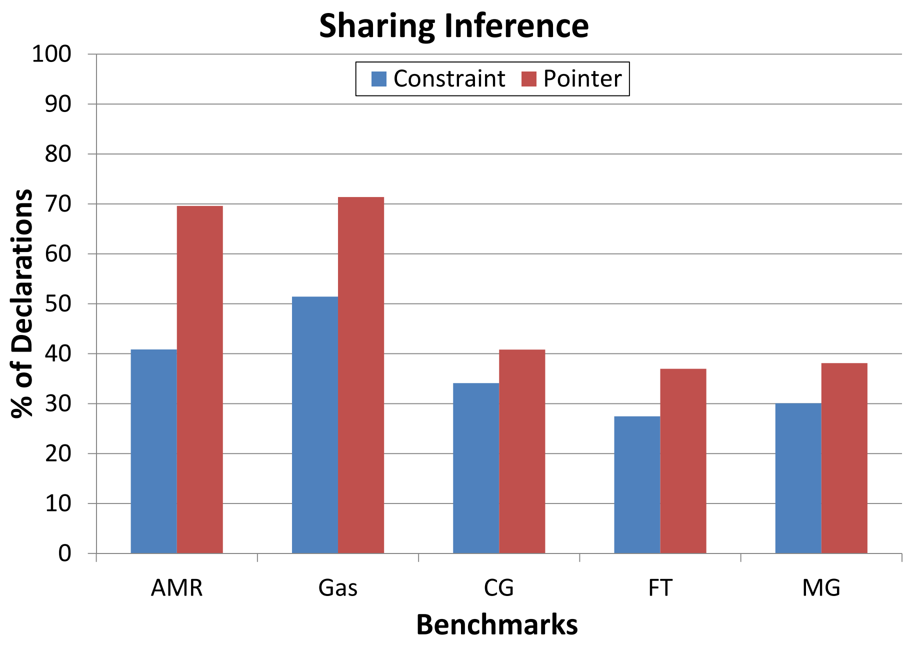


Figure 8.35: Sharing inference results for some Titanium programs.

$$w_{min} = max\{n \mid \exists a = (l, n) \in S\}$$

A reference is *local* if it can only be to the same physical address space as the source thread. In the two-level pointer analysis, a reference is local if its minimal width is 1, while in the three-level analysis, it is local if its minimal width is 2.

We have implemented locality inference using pointer analysis in the Titanium compiler. Figure 8.34 compares the precision of this inference to Liblit and Aiken's constraint-based local qualification inference (LQI) [65] on the reachable local variables, fields, and method parameters and return values of reference type in the application code. Since the pointer analysis distinguishes between allocation sites, the three-level pointer analysis versions perform better than LQI. In most of the benchmarks, the two-level analysis performs worse than LQI, since it cannot take into account casts to **`local`**, which are used often in the benchmarks to manually eliminate runtime locality checks.

8.3.2 Sharing Inference

An object in a parallel program is *private* if it is never leaked beyond its source thread. A reference is private if it can only refer to private objects. As described in §6.8.2.2, our pointer analysis implementation only creates wide versions of an abstract location if that location can be leaked. Thus, an abstract location is private if it has no wide counterparts, and a variable or expression is private if it evaluates to an abstract set that only contains private locations. This inference is

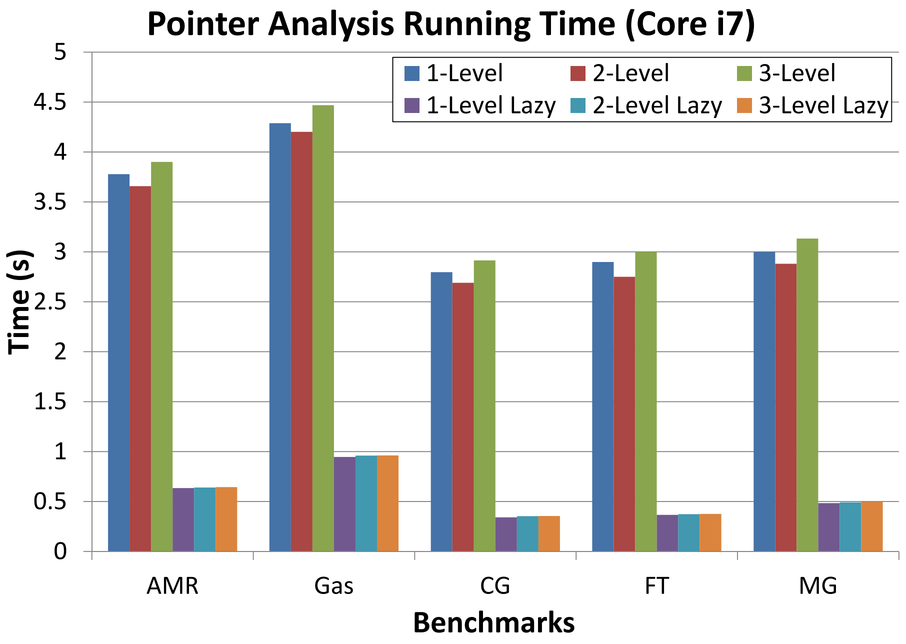


Figure 8.36: Analysis running time results for some Titanium programs, using various levels of pointer analysis.

independent of the number of levels in the analysis hierarchy, as long as at least level 1 is separate from the rest of the levels.

We have implemented data-sharing inference in the Titanium compiler using pointer analysis. The precision of this inference compared to Liblit, Aiken, and Yelick's constraint-based sharing qualification inference (SQI) [66] is shown in Figure 8.35. As with locality inference, results are only reported for reachable local variables, fields, and method parameters and return values of reference type in the application code. Again, the pointer analysis version does considerably better than the constraint-based version, since it separates allocation sites.

8.3.3 Performance

Though our implementation is not as optimized as possible, its performance still demonstrates some interesting results. Figure 8.36 shows the running time of various levels of pointer analysis on a 2.93 GHz Core i7 machine, with and without the lazy analysis optimization described in §6.8.2.1. The optimization is very effective, decreasing execution time by an average of almost 85% for the benchmarks above. The performance difference between one, two, and three levels of hierarchy is nonexistent, with all three averaging 0.56 seconds for the same set of programs using the lazy analysis optimization. This validates our decision to allow an arbitrary number of levels in the analysis, since execution time would increase linearly with the number of levels if a two-level analysis was used multiple times instead.

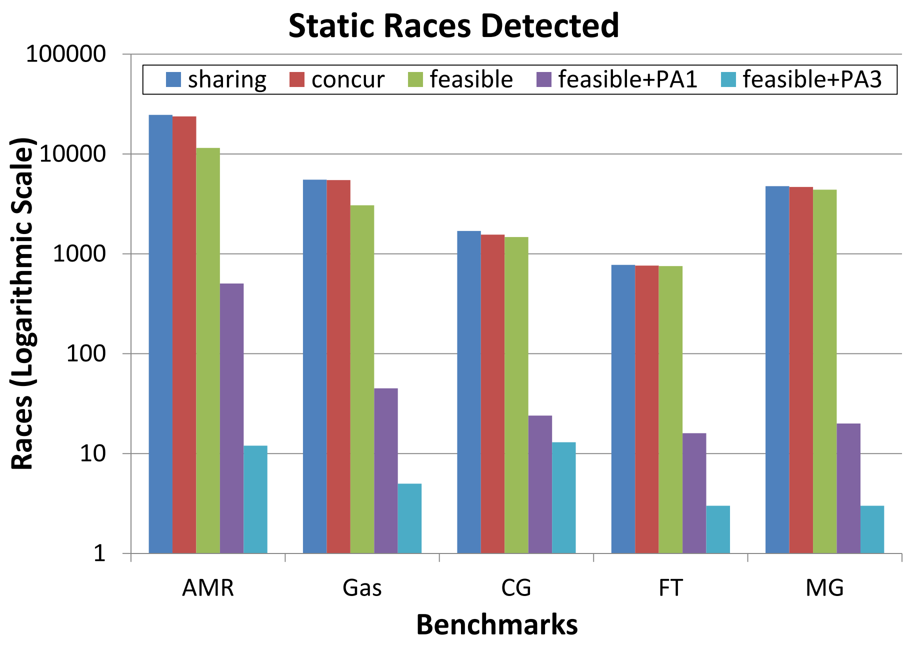


Figure 8.37: Number of data races detected at compile-time.

8.4 Combined Analyses

We evaluate the combination of our pointer and concurrency analyses using two clients: static race detection and enforcing sequential consistency at the language/compiler level. We use the same set of benchmarks as in §8.3 for our evaluation.

8.4.1 Static Race Detection

Using our concurrency and pointer analyses, we built a compile-time data race analysis into the Titanium compiler. Static information is generally not enough to determine with certainty that two memory accesses compose a race, so nearly all reported races are false positives. (The correctness of the concurrency and pointer analyses ensure that no false negatives occur.) We therefore consider a race detector that reports the fewest races to be the most effective.

Figure 8.37 compares the effectiveness of five levels of race detection:

- **sharing**: Type-based alias analysis and Liblit and Aiken's constraint-based sharing inference [66] are used to detect potential races.
- **concur**: Our basic concurrency analysis (§7.1.2) is used to eliminate non-concurrent races.
- **feasible**: Our feasible-paths concurrency analysis (§7.1.3) is used to eliminate non-concurrent races.

- **feasible+PA1**: A single-level pointer analysis is used to eliminate false aliases.
- **feasible+PA3**: A three-level pointer analysis is used to eliminate false aliases.

The results show that the concurrency and pointer analyses can eliminate most of the races reported by our detector. None of the benchmarks benefit significantly from the basic concurrency analysis, but the feasible-paths version significantly reduces the number of races found in two of the benchmarks. The addition of pointer analysis removes most of the remaining races, with a three-level analysis providing significant benefits over a one-level analysis. The small number of remaining potential races allowed us to find an actual bug in the conjugate gradient application.

8.4.1.1 Discussion of Conjugate Gradient

In §8.2, §8.3, and §8.4.1, we used a version of the conjugate gradient benchmark that predates the addition of teams. As noted in §8.1.2, this version uses hand-written reductions over matrix rows. There are actually two variants of this code. The first uses global barriers for synchronization, as follows.

```
// store data in index -(i+1) of allResults array on remote proc
allResults[targetProc[i]].slice(1,-(i+1)).copy(myResults.slice(1,i));
// wait for other procs to finish writing
Ti.barrier();
// process new data
...
```

The second variant uses point-to-point write flags with spin locks.

```
1  // store data in index -(i+1) of allResults array on remote proc
2  allResults[targetProc[i]].slice(1,-(i+1)).copy(myResults.slice(1,i));
3  // write flag on remote proc announcing the comm. is complete
4  allWriteFlags[targetProc[i]][i] = 1;
5  // see if other remote proc has completed writing to my array
6  while (myWriteFlags[i] == 0) {
7    Ti.poll();
8  }
9  // process new data
10 ...
```

The latter variant is actually incorrect under Titanium's relaxed memory-consistency model, as the writes on lines 2 and 4 may be reordered.

Unfortunately, the incorrect point-to-point synchronization scheme is the default variant used in the original CG implementation. As a result, all our experiments in the past three sections use this variant. In race detection, this variant results in two more detected races than the barrier variant: one race condition between writing data in line 2 and reading it in line 10, and one race between writing flags in line 4 and reading them in line 6. Unlike most of the other races detected, these are actually valid races that can result in incorrect execution under a relaxed memory model.

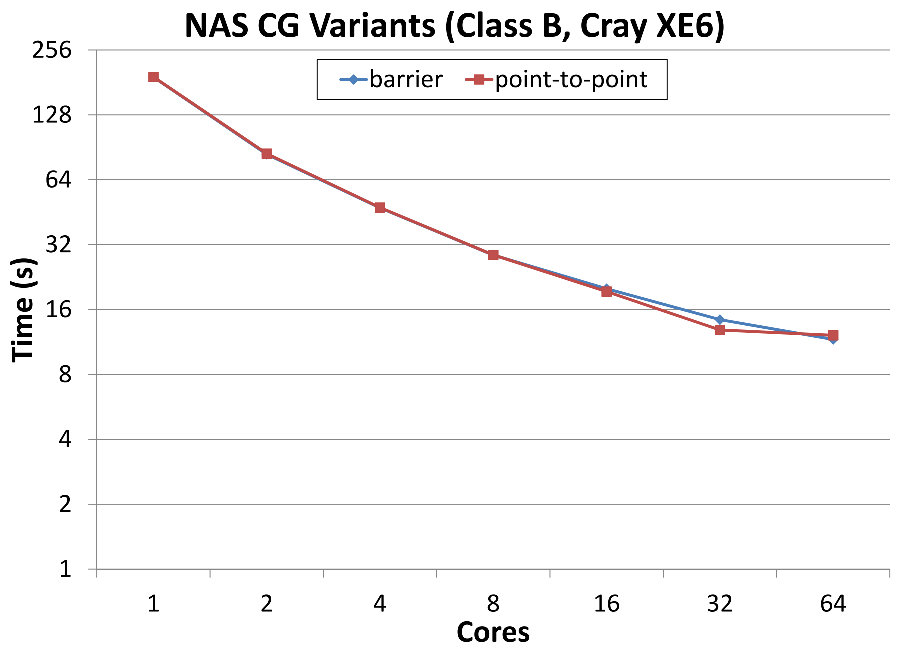


Figure 8.38: Performance of barrier and point-to-point variants of CG on a Cray XE6.

As part of our experiments in §8.1.2, we compared the two variants and found that 80% of our runs failed to produce correct results on 64 threads in the point-to-point case, while all runs succeeded in the barrier case. Figure 8.38 shows that the two variants offer similar performance[3]. As a result, we used the barrier variant in §8.1.2 to compare performance to team-based versions of the code and in §8.4.2.

This illustrates an important lesson: the underlying language, libraries, and runtime system must provide sufficient tools for the programmer to implement an algorithm without having to resort to potentially incorrect code. In this case, it was the lack of teams and team collectives that forced the programmer to implement an incorrect hand-written version.

8.4.2 Sequential Consistency

Sequential consistency is enforced by inserting *memory barriers*, also called *memory fences*, into a program. These memory barriers prevent the compiler from reordering code and the runtime from reordering memory accesses. As a result, they can negatively affect performance, preventing optimizations both at compile-time and during execution. The number of memory barriers inserted by the compiler, which we refer to as *static fences*, roughly corresponds to the amount of optimization prevented in the compiler. However, many of these memory barriers may be in unreachable or non-critical code, so the number of fences actually executed at runtime, or *dynamic*

[3]Absolute performance is somewhat slower than reported in §8.1.2, since the experiments here were run on an older, less optimized version of the code.

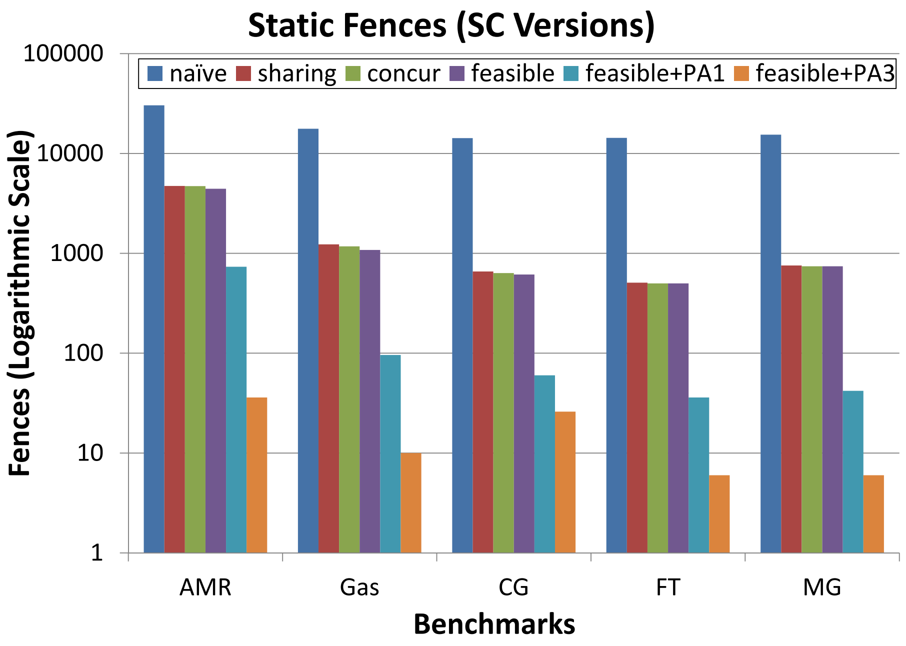


Figure 8.39: Number of memory barriers generated at compile-time.

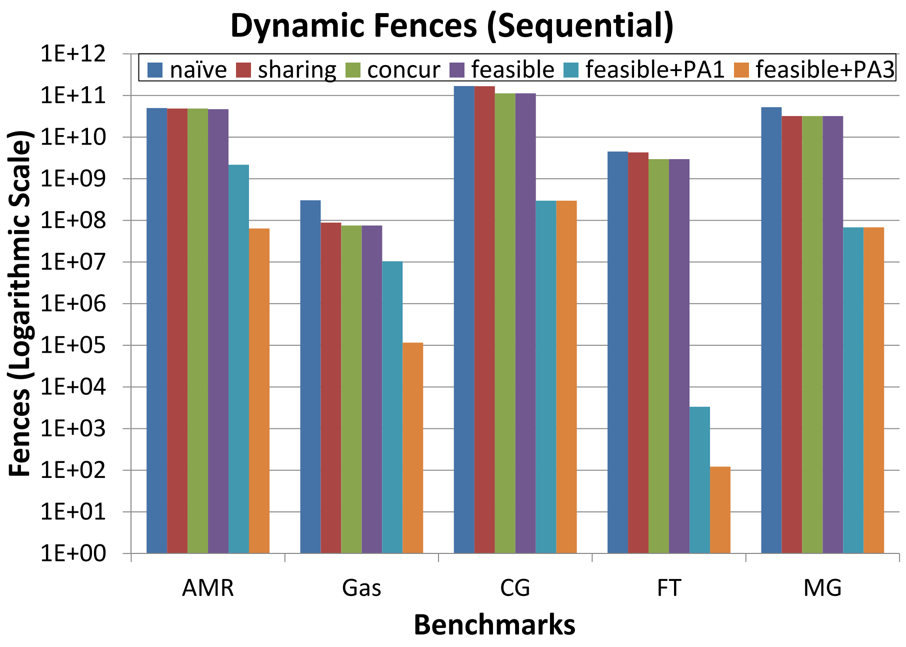


Figure 8.40: Average number of memory barriers executed at runtime. Benchmarks were run on a Core i7 using a single thread.

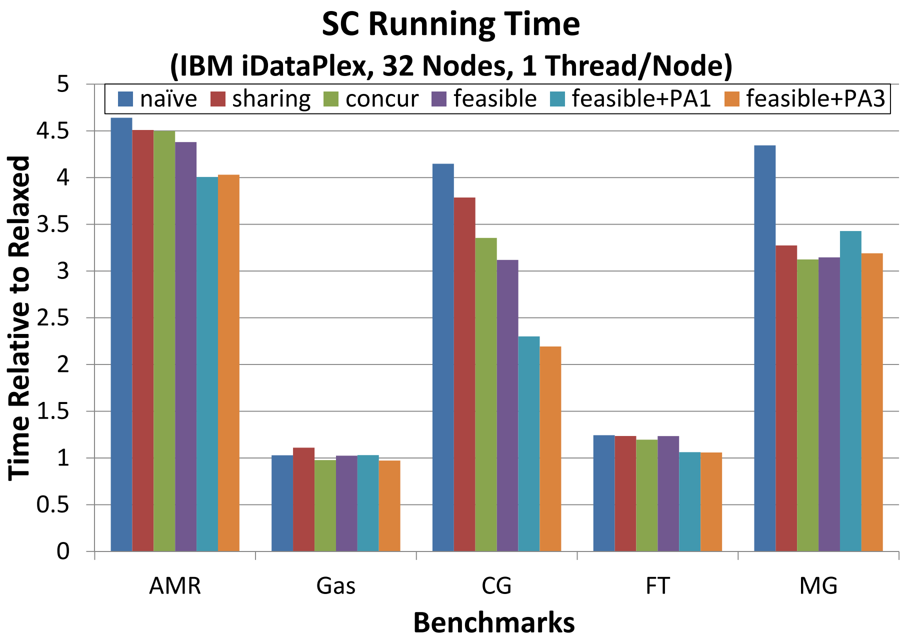


Figure 8.41: Execution time on an IBM iDataPlex using 32 processors on 32 nodes, compared to a relaxed-consistency version of the code.

fences, provides a better estimate of the performance impact of sequential consistency. Finally, we measure the actual running time of each benchmark on two platforms, the IBM iDataPlex and the Cray XE6 described in §8.1.1. We tested distributed performance on the IBM machine, using 32 processors on 32 nodes. On the Cray XE6, we used four processors on a single NUMA node to test shared-memory performance. We compared execution time of sequentially-consistent versions of each benchmark to a version compiled using Titanium's default relaxed memory model.

Figure 8.39 compares the number of memory barriers generated for each program using the five different levels of analysis in §8.4.1, with an additional base level of analysis:

- **naïve**: Fences are inserted around all heap accesses.

Figure 8.40 compares the resulting dynamic counts at runtime, and Figure 8.41 shows the execution time of each sequentially-consistent version, relative to the relaxed version, on the distributed machine. On the shared-memory machine, we found that the four lower levels of analysis resulted in code that runs multiple orders of magnitude slower than the relaxed version. As a result, we were only able to collect data from the two sequentially-consistent versions with pointer analysis, and Figure 8.42 shows the results.

The results show that our analysis, at its highest precision, is effective in reducing the numbers of both static and dynamic memory barriers. The execution time, however, is less affected by the analyses. Running time for Gas and FT nearly match the relaxed versions on the distributed machine for all levels of analysis. AMR shows a small benefit and CG a larger benefit from

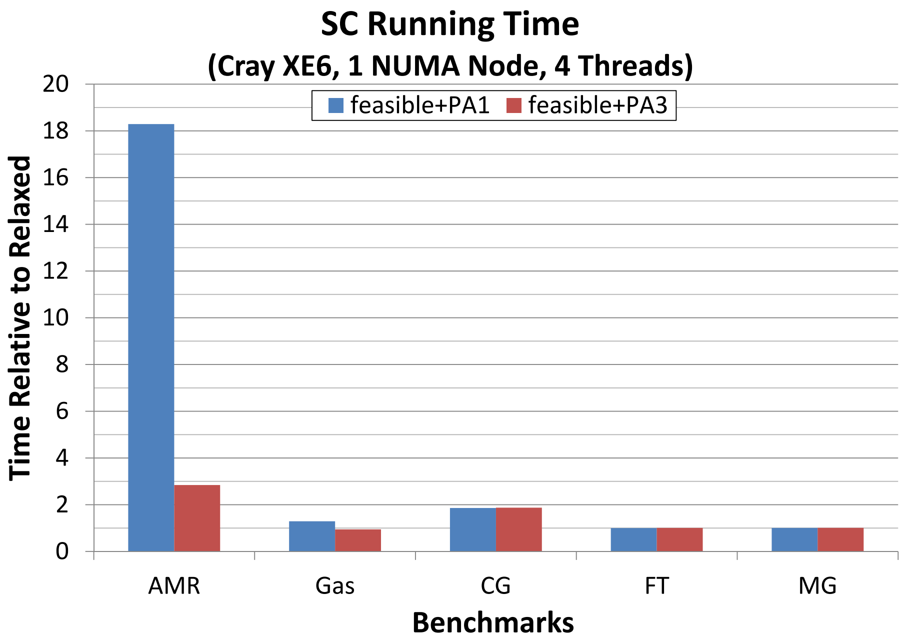


Figure 8.42: Execution time on a Cray XE6 using four processors on a single NUMA node, compared to a relaxed-consistency version of the code.

pointer analysis. In MG, neither concurrency nor pointer analysis improves execution time over constraint-based sharing inference. Overall, the benchmarks average 2.3 times the running time of their respective relaxed versions at the highest level of analysis.

On the shared-memory machine, execution time is somewhat better at the most precise level of analysis, averaging 1.5 times the relaxed versions. AMR shows a very large benefit from multi-level pointer analysis over flat pointer analysis, and GAS also shows a small benefit, matching the performance of the relaxed version. CG does not benefit from the three-level hierarchical pointer analysis, while FT and MG match relaxed performance in both sequentially-consistent versions.

Chapter 9

Related Work

Parallel computation has been an active area of research for decades, so there is a wide body of work related to the concepts in this dissertation. In this chapter, we review some of the work that has been done in the areas of hierarchical computation, collective alignment, concurrency analysis, race detection, and sequential consistency. This is not meant to be an exhaustive review of the work in these areas; rather, we discuss a subset of the related work in order to provide some context to our own efforts in this thesis.

9.1 Hierarchical Computation

In §2.3, we noted that flat Titanium exposes three levels of machine hierarchy in its memory model while Unified Parallel C (UPC) exposes two levels. Many other languages also have two-level memory models, including languages that do not use the single program, multiple data (SPMD) model of parallelism. In the X10 language [88], the memory and space is composed of *places*, and tasks execute at specific places. Remote data can only be accessed by spawning a task at the target place. Chapel [25] has a similar concept of *locales*, and it allows data structures to be distributed across locales. Data-parallel operations over such data structures spawn tasks at each locale to locally operate on data, and tasks can also be spawned at particular locales.

Only a handful of existing parallel languages incorporate hierarchical programming constructs beyond two levels of hierarchy.

In the Fortress language [4], memory is divided into an arbitrary hierarchy of *regions*. Data structures can be spread across multiple regions, and tasks can be placed in particular regions by the programmer.

The Sequoia project [36] incorporates machine hierarchy in its language model. A Sequoia program consists of a hierarchy of tasks that get mapped to the computational units in a hierarchical machine. Sequoia has two types of tasks: *inner tasks* that decompose computations into subtasks and *leaf tasks* that perform actual computation. Both the height and width of the resulting task hierarchy can be controlled by the user when starting the program. Communication between tasks is very limited: only parent and child tasks can communicate, through the use of parameters. This restriction on communication as well as the lack of collective operations make the Sequoia model unsuitable for many applications written in SPMD and PGAS languages.

The hierarchical place trees (HPT) abstraction [106] extends the Sequoia model to allow more general communication between tasks and incorporates X10's ability to spawn tasks at specific locations in a machine. The programming model, however, is still essentially task parallel, with task queues at each location to run tasks. This model both lacks the simple, analyzable structure of SPMD parallelism and the latter's mechanisms for cooperative synchronization and communication.

Hierarchically tiled arrays (HTAs) [11] allow data structures to be hierarchically decomposed to match a target machine's layout. A program can then operate on these data structures in an essentially data-parallel manner, with the compiler and runtime mapping execution according to the data layout. Like other data-parallel languages, however, the HTA model is quite restrictive, as it is difficult to write applications with irregular task or communication structures.

Nested data parallelism allows hierarchical algorithms to be expressed in the context of data parallelism. The model has been implemented in NESL [41, 15, 13, 14] and in Haskell [22, 21, 46]. However, as mentioned above, irregular algorithms can be difficult to express in the data parallel model, and nested data-parallel implementations have focused on vector and shared-memory machines rather than hierarchical machines. They also require more complicated compilers than SPMD languages.

Much recent work has been done on hierarchical load balancing for parallel applications. Zheng et al. demonstrated [113] such an approach in the context of Charm++ [47], a language with a concurrent object-oriented model. They divide the set of processors into independent groups arranged in a hierarchy. Runtime measurements are then used to balance load across each subtree in the hierarchy. The HotSLAW library for UPC [74] extends the SPMD model of UPC with dynamic task parallelism. Dynamically created tasks are executed using task queues on a subset of the UPC threads, and a hierarchical work-stealing algorithm balances load according to user-defined, hierarchical locality domains.

The hierarchical single program, multiple data (HSPMD) model is in some sense the inverse of the RSPMD model. In RSPMD, an initial, fixed set of threads is recursively subdivided into smaller teams of cooperating threads. In HSPMD, on the other hand, there is only a single thread initially, and each thread can spawn a new set of cooperating threads. The Phalanx programming model uses a version of HSPMD [38]. We considered the HSPMD model as well [50], but concluded that it requires a more complicated compiler and runtime implementation than RSPMD.

Various analytical models for computation on hierarchical machines have been studied. The Multi-BSP model [101] consists of a hierarchical set of memory spaces, with processors at the lowest level. Each level has parameters for memory size, communication cost, and synchronization cost. Using this model, optimal algorithms can be defined for problems such as matrix multiplication and fast Fourier transform. The D-BSP [98, 12] model also incorporates hierarchical communication costs, but unlike Multi-BSP, it does not model memory and cache hierarchies. Other models encompass multiprocessor memory hierarchies but not hierarchical communication costs [102, 89].

The concept of thread teams has been gaining popularity in the SPMD community. MPI has communicators that allow a subset of threads to perform collective operations, and other communication layers and programming languages have recently introduced or are in the process of introducing similar team constructs. However, MPI communicators place no restriction on the underlying thread structure of a team, and a thread can be a part of multiple communicators concurrently, making it easy to deadlock a program through improper use of communicators. Even correct use of multiple communicators can be difficult for programmers to understand and compilers to analyze, as they must reason about the order of communicator calls on each thread. Finally, communicators do not have a hierarchical structure, so they cannot easily reflect the layout of the underlying machine.

9.2 Collective Alignment

In addition to Aiken and Gay's work on structural correctness, many others have addressed the problem of collective alignment.

Zhang and Duesterwald developed a static analysis for matching textually-unaligned barriers in MPI [111]. The analysis is available as part of the Eclipse Parallel Tools Platform [33]. They later extended their analysis in collaboration with Gao to shared-memory OpenMP programs and built a concurrency analysis on top of it [112]. Siegel and Avrunin applied model checking to MPI programs [92]. Their system detects deadlock in the presence of MPI barriers.

Jeremiassen and Eggers developed a static analysis for barrier synchronization for SPMD programs with non-textual barriers that divides a program into non-concurrent phases [45]. This analysis provides a conservative estimate of which barriers can be erroneously aligned: a barrier can only be misaligned if it and another barrier may both be executed at the boundary of the same phase. Other work has also been done in the area of concurrency analysis [60, 67], though like Jeremiassen and Eggers, the authors don't directly apply it to detecting alignment errors.

A lot of work has also been done in the area of barrier optimization [26, 78, 99], which requires reasoning about the execution of barriers. However, work in this area generally either assumes aligned barriers or is not concerned with detecting synchronization errors arising from the misaligned barriers.

The main drawback to static analysis is imprecision: it may be unable to conclude that a collective is properly aligned even if it is. While in practice, results from static analysis appear to be precise enough to be useful for other analyses and optimizations, the existence of false positives makes it less than ideal for enforcing semantic restrictions, as it would report nonexistent errors to the programmer.

9.3 Pointer Analysis

The language and type system we presented here are generalizations of those described by Liblit and Aiken [65]. They defined a two-level hierarchy and used it to produce a constraint-based analysis that infers locality information about pointers. Later with Yelick, they extended the language and type system to consider sharing of data, and they defined another constraint-based analysis to infer sharing properties of pointers [66].

Pointer analysis was first described by Emami [34] and Andersen [6], and later extended by others to parallel programs. Rugina and Rinard developed a thread-aware pointer analysis for the Cilk multithreaded programming language [87] that is both flow-sensitive and context-sensitive. Others such as Zhu and Hendren [114] and Hicks [42] have developed flow-insensitive versions for multithreaded languages. However, none of these analyses consider hierarchical, distributed machines.

The pointer analysis we presented here is a generalization and formalization of the analysis sketched in a previous paper [51]. That analysis is similar to a two-level version of our hierarchical analysis, but the abstraction is quite different. Only the abstraction of the `transmit` operation was described in that paper, though an almost complete implementation was done.

9.4 Concurrency Analysis

An extensive amount of work on concurrency analysis has been done for both languages with dynamic parallelism and SPMD programs. Duesterwald and Soffa presented a data-flow analysis to compute the *happened-before* and *happened-after* relation for program statements [32]. Their analysis is for detecting races in programs based on the Ada rendezvous model [100]. Masticola and Ryder developed a more precise non-concurrency analysis for the same set of programs [68]. The results are used for debugging and optimization. Jeremiassen and Eggers developed a static analysis for barrier synchronization for SPMD programs with non-textual barriers [45]. They used the information to reduce false sharing on cache-coherent machines.

9.5 Race Detection

Others besides Duesterwald and Soffa and Masticola and Ryder have developed tools for race detection. Flanagan and Freund presented a static race-detection tool for Java based on type inference and checking [37]. Boyapati and Rinard developed a type system for Java that guarantees that a program is race-free [18]. Tools such as Eraser [90] and TRaDe [24] detect races at runtime instead of statically. Our prior work presented a concurrency-analysis algorithm and race detection for Titanium [52]. Other static and dynamic race-detection schemes have also been developed [85, 8, 31, 23, 79].

9.6 Sequential Consistency

The memory consistency issue arises in a language with an explicitly parallel semantics and some type of shared address space. The class of such languages includes Java, UPC, Titanium, and Co-Array Fortran, some of the languages proposed in the recent HPCS effort, as well as shared-memory language extensions such as POSIX Threads and OpenMP [19, 77, 80, 110].

Shasha and Snir provided some of the foundational work in enforcing sequential consistency from a compiler level when they introduced the idea of *cycle detection* [91]. However, that work was designed for general MIMD parallelism, limited to straight-line code, and was not designed as a practical static analysis. Midkiff and Padua outlined some of the implementation techniques that could violate sequential consistency and developed some static analysis ideas, including a concurrent static single assignment form in a paper by Lee et al. [63]. As part of the Pensieve project, Lee and Padua exploited properties of fences and synchronization to reduce the number of delays in cycle detection [64]. The project also includes a Java compiler that takes a memory model as input [95]. More recently, Sura et al. have shown that cooperating escape, thread structure, and delay set analyses can be used to provide sequential consistency cheaply in Java [96]. Our work differs from theirs in two primary ways: 1) we take advantage of some of the synchronization paradigms, such as barriers, that exist in SPMD programs, and 2) our machine targets include distributed-memory architectures where the cost of a memory fence is essentially that of a round-trip communication across the network.

The earliest implementation work on cycle detection was by Krishnamurthy and Yelick for the restricted case of SPMD programs [60]. That was done in a simplified subset of the Split-C language and introduced a polynomial-time algorithm for cycle detection in SPMD programs. They also used synchronization analysis to reduce the number of fences, but their source language did not have the restriction that barriers must match textually and they did not take advantage of single conditionals. At compile time, they generated two versions of the code, one assuming the barriers line up and the other one not. At runtime, they switched between the two versions depending on how the barriers were executed. Our approach does not suffer the same runtime overhead and code bloat that exists in theirs. In addition, their compiler used only a simple type-based alias analysis.

There has also been work done in the area of reducing the number of fences required to enforce sequential consistency. Liblit, Aiken, and Yelick developed a type system to identify shared data accesses in Titanium programs [66], and for sequential consistency, they only insert a fence at each shared data access identified. Based on our experimental results in §8, our technique is a significant improvement over theirs in terms of static fence count, dynamic fence count, and running time of the generated programs.

Chapter 10

Conclusion

Achieving good performance on modern machines has become much harder since the advent of multicore processors. It will only become more difficult as machines become more hierarchical, resulting in a wide array of communication costs between processing cores. In order to make the task tractable, new programming models, libraries, and runtime systems are needed to allow programmers to express hierarchical computation with minimal effort.

While many approaches to the hierarchy problem are possible, including various degrees of mixing data and task parallelism, we chose to base ours on the model of single program, multiple data (SPMD). In this model, all threads execute the same code, and powerful collective operations can be used for synchronizing and communicating among threads. As a result of its simple structure, SPMD has proven to provide good performance, productivity, safety, and analyzability.

We defined the recursive single program, multiple data (RSPMD) model that extends SPMD with hierarchical, structured teams. We designed RSPMD extensions to the Titanium language, showing that the combination of a team data structure and lexical usage constructs prevents erroneous usage of teams. Team collectives bring the performance and productivity of collective operations to subsets of threads in a program.

A common problem in the SPMD model is ensuring that collective operations are aligned so that all threads execute the same sequence of collectives, avoiding deadlock. We demonstrated how to ensure alignment of global collectives through dynamic checking, reducing programmer burden over static schemes. We extended this system to team collectives, providing important safety guarantees in RSPMD programs.

The simple structure of SPMD programs and collective operations enable precise but efficient analyses. We presented a hierarchical pointer analysis for RSPMD programs that infers where each pointer's data is allocated and on which threads it is located. We showed that this analysis improves precision of locality and sharing inference, which allow many program optimizations to be performed.

We also defined a global concurrency analysis for SPMD programs, taking advantage of aligned collectives. Such an analysis determines the set of concurrent statements in a program, information important to many other optimizations and analyses such as race detection. We showed how to improve precision of the analysis by eliminating impossible program paths, without degrading the theoretical efficiency of the analysis. We proceeded to demonstrate how to extend this analysis to compute concurrency information at the granularity of individual teams.

We combined our pointer and concurrency analyses to detect race conditions, a common type of parallel bug, and to enforce a sequentially-consistent memory model that is easier for users to understand than alternative models. In race detection, our analyses reduced the number of false positives by three orders of magnitude, enabling us to find an actual bug in a conjugate gradient application. The analyses also lowered the performance cost of providing sequential consistency to within a factor of four on a distributed machine and three on a shared-memory platform.

We implemented multiple benchmarks using the RSPMD model, including sorting, conjugate gradient, particle in cell, and stencil. We demonstrated that hierarchical teams enable divide-and-conquer algorithms such as sorting to be implemented elegantly. We showed that team collectives provide better performance than hand-written alternatives in conjugate gradient, resulting in code that runs nearly twice as fast at high thread counts. We also demonstrated that hierarchical teams enable optimizations for hierarchical machines to be written in the context of a single programming model. These optimizations enabled the particle in cell benchmark to scale to eight times as many threads as the non-hierarchical version and increased performance of sorting by up to 1.4x. We showed that our hierarchical model beats the standard mechanism of combining a distributed library with a shared-memory one by as much as 14% in stencil.

While we demonstrated the benefits of RSPMD for various benchmarks, challenges do remain in the area of hierarchical computation. Our experiments with conjugate gradient and particle in cell showed that shared memory cannot always be exploited for performance gains. The latter benchmark also illustrated the increased tuning space of hierarchical codes; while this introduces new opportunities for optimization, it can also require more programmer effort to achieve maximal performance. In addition, the stencil code proved the difficulty of composing different parallel libraries. Finally, we did not address other important problems in large-scale parallel computing, such as heterogeneity and fault tolerance.

However, we did show that RSPMD provides safety, analyzability, expressiveness, and performance on hierarchical machines. We believe that the model is a significant step towards making parallel programming easier for users. We also believe that it provides a useful building block for further simplifying parallel programming, as it can be applied in libraries and specializers to hide most of the difficult details of parallelism from end users.

Bibliography

[1] Alexander Aiken and David Gay. "Barrier Inference". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 1998.

[2] Alexander Aiken and David Gay. "Memory Management with Explicit Regions". In: *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. Montreal, Canada, 1998.

[3] Jonathan Aldrich et al. "Comprehensive Synchronization Elimination for Java". In: *Science of Computer Programming* 47 (2002).

[4] Eric Allen et al. *The Fortress Language Specification, Version 0.866*. Sun Microsystem Inc. Feb. 13.

[5] George Almási et al. "UPC Collectives Library 2.0". In: *Fifth Conference on Partitioned Global Address Space Programming Models*. Galveston Island, Texas, 2011.

[6] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, 1994.

[7] Applied Numerical Algorithms Group (ANAG). *Chombo*. Lawrence Berkeley National Laboratory. URL: http://seesar.lbl.gov/ANAG/software.html.

[8] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. "Guava: a Dialect of Java without Data Races". In: *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Minneapolis, Minnesota, United States: ACM Press, 2000, pp. 382–400. ISBN: 1-58113-200-X. DOI: http://doi.acm.org/10.1145/353171.353197.

[9] David Bailey et al. "The NAS Parallel Benchmarks". In: *The International Journal of Supercomputer Applications* 5.3 (1991), pp. 63–73. URL: http://citeseer.nj.nec.com/article/bailey94nas.html.

[10] Marsha Berger and Phillip Colella. "Local Adaptive Mesh Refinement for Shock Hydrodynamics". In: *Journal of Computational Physics* 82.1 (1989). Lawrence Livermore Laboratory Report No. UCRL-97196, pp. 64–84.

[11] Ganesh Bikshandi et al. “Programming for Parallelism and Locality with Hierarchically Tiled Arrays”. In: *PPoPP ’06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, New York, USA: ACM, 2006, pp. 48–57. ISBN: 1-59593-189-9. DOI: http://doi.acm.org/10.1145/1122971.1122981.

[12] Gianfranco Bilardi et al. “On the Effectiveness of D-BSP as a Bridging Model of Parallel Computation”. In: *Proceedings of the International Conference on Computational Science-Part II*. ICCS ’01. London, UK, UK: Springer-Verlag, 2001, pp. 579–588. ISBN: 3-540-42233-1. URL: http://dl.acm.org/citation.cfm?id=645456.654690.

[13] Guy E. Blelloch. “Programming Parallel Algorithms”. In: *Commun. ACM* 39.3 (Mar. 1996), pp. 85–97. ISSN: 0001-0782. DOI: 10.1145/227234.227246. URL: http://doi.acm.org/10.1145/227234.227246.

[14] Guy E. Blelloch and John Greiner. “A Provable Time and Space Efficient Implementation of NESL”. In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP ’96. Philadelphia, Pennsylvania, United States: ACM, 1996, pp. 213–225. ISBN: 0-89791-770-7. DOI: 10.1145/232627.232650. URL: http://doi.acm.org/10.1145/232627.232650.

[15] Guy E. Blelloch et al. “Implementation of a Portable Nested Data-Parallel Language”. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’93. San Diego, California, United States: ACM, 1993, pp. 102–111. ISBN: 0-89791-589-5. DOI: 10.1145/155332.155343. URL: http://doi.acm.org/10.1145/155332.155343.

[16] Jeff Bogda and Urs Hölzle. “Removing Unnecessary Synchronization in Java”. In: OOPSLA ’99 (1999), pp. 35–46. DOI: 10.1145/320384.320388. URL: http://doi.acm.org/10.1145/320384.320388.

[17] Dan Bonachea. *GASNet Specification, v1.1*. Tech. rep. UCB/CSD-02-1207. University of California, Berkeley, 2002.

[18] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. “Ownership Types for Safe Programming: Preventing Data Races and Deadlocks”. In: *OOPSLA ’02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Seattle, Washington, USA: ACM Press, 2002, pp. 211–230. ISBN: 1-58113-471-1. DOI: http://doi.acm.org/10.1145/582419.582440.

[19] William Carlson et al. *Introduction to UPC and Language Specification*. Tech. rep. CCS-TR-99-157. IDA Center for Computing Sciences, 1999.

[20] Bryan Catanzaro et al. *SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization*. Tech. rep. UCB/EECS-2010-23. University of California at Berkeley, 2010.

[21] Manuel M. T. Chakravarty et al. "Data Parallel Haskell: A Status Report". In: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. DAMP '07. Nice, France: ACM, 2007, pp. 10–18. ISBN: 978-1-59593-690-5. DOI: 10.1145/1248648.1248652. URL: http://doi.acm.org/10.1145/1248648.1248652.

[22] Manuel M. T. Chakravarty et al. "Nepal - Nested Data Parallelism in Haskell". In: *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. Euro-Par '01. London, UK, UK: Springer-Verlag, 2001, pp. 524–534. ISBN: 3-540-42495-4. URL: http://dl.acm.org/citation.cfm?id=646666.699740.

[23] Guang-Ien Cheng et al. "Detecting Data Races in Cilk Programs that Use Locks". In: *SPAA '98: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. Puerto Vallarta, Mexico: ACM Press, 1998, pp. 298–309. ISBN: 0-89791-989-0. DOI: http://doi.acm.org/10.1145/277651.277696.

[24] Mark Christiaens and Koen De Bosschere. "TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs". In: *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*. 2001.

[25] Cray Inc. *Chapel Specification 0.4*. Feb. 4.

[26] Alain Darte and Robert Schreiber. "A Linear-Time Algorithm for Optimal Barrier Placement". In: *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Chicago, IL, USA: ACM, 2005, pp. 26–35. ISBN: 1-59593-080-9. DOI: http://doi.acm.org/10.1145/1065944.1065949.

[27] Kaushik Datta. "Auto-tuning Stencil Codes for Cache-Based Multicore Platforms". PhD thesis. University of California, Berkeley, 2009.

[28] Kaushik Datta. "The NAS Parallel Benchmarks in Titanium". MA thesis. University of California, Berkeley, 2005.

[29] Kaushik Datta, Dan Bonachea, and Katherine Yelick. "Titanium Performance and Potential: an NPB Experimental Study". In: *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. 2005.

[30] Kaushik Datta et al. "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors". In: *SIAM Rev.* 51.1 (Feb. 2009), pp. 129–159. ISSN: 0036-1445. DOI: 10.1137/070693199. URL: http://dx.doi.org/10.1137/070693199.

[31] Anne Dinning and Edith Schonberg. "Detecting Access Anomalies in Programs with Critical Sections". In: *PADD '91: Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*. Santa Cruz, California, United States: ACM Press, 1991, pp. 85–96. ISBN: 0-89791-457-0. DOI: http://doi.acm.org/10.1145/122759.122767.

[32] Evelyn Duesterwald and Mary Lou Soffa. “Concurrency Analysis in the Presence of Procedures Using a Data-Flow Framework”. In: *Symposium on Testing, Analysis, and Verification*. Victoria, British Columbia, 1991.

[33] *Eclipse Parallel Tools Platform*. URL: http://www.eclipse.org/ptp/.

[34] Maryam Emami. “A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing Compiler”. MA thesis. McGill University, Montreal, 1993.

[35] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. “A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics”. In: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*. STOC ’03. San Diego, CA, USA: ACM, 2003, pp. 448–455. ISBN: 1-58113-674-9. DOI: 10.1145/780542.780608. URL: http://doi.acm.org/10.1145/780542.780608.

[36] Kayvon Fatahalian et al. “Sequoia: Programming the Memory Hierarchy”. In: 2006, pp. 4–4. DOI: 10.1109/SC.2006.55.

[37] Cormac Flanagan and Stephen N. Freund. “Type-Based Race Detection for Java”. In: *PLDI ’00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 219–232. ISBN: 1-58113-199-2. DOI: http://doi.acm.org/10.1145/349299.349328.

[38] Michael Garland, Manjunath Kudlur, and Yili Zheng. “Designing a Unified Programming Model for Heterogeneous Machines”. In: To appear in *Supercomputing 2012*. 2012.

[39] David Gay. “Barrier Inference”. PhD thesis. University of California, Berkeley, 1998.

[40] David Gay and Alexander Aiken. “Language Support for Regions”. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. Snowbird, Utah, 2001.

[41] Blelloch Guy E. *NESL: A Nested Data-Parallel Language (3.1)*. Tech. rep. CMU-CS-95-170. Carnegie Mellon University, 1995.

[42] James Hicks. “Experiences with Compiler-Directed Storage Reclamation”. In: *FPCA ’93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Copenhagen, Denmark: ACM Press, 1993, pp. 95–105. ISBN: 0-89791-595-X. DOI: http://doi.acm.org/10.1145/165180.165194.

[43] Paul Hilfinger et al. *Titanium Language Reference Manual*. Tech. rep. UCB/EECS-2005-15. University of California, Berkeley, 2005.

[44] J.S. Huang and Y.C. Chow. “Parallel Sorting and Data Partitioning by Sampling”. In: *7th International Computer Software and Applications Conference*. 1983.

[45] Tor Jeremiassen and Susan Eggers. “Static Analysis of Barrier Synchronization in Explicitly Parallel Programs”. In: *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*. Montreal, Canada, 1994.

[46] Simon Peyton Jones et al. “Harnessing the Multicores: Nested Data Parallelism in Haskell”. In: *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. APLAS ’08. Bangalore, India: Springer-Verlag, 2008, pp. 138–138. ISBN: 978-3-540-89329-5. DOI: 10.1007/978-3-540-89330-1_10. URL: http://dx.doi.org/10.1007/978-3-540-89330-1_10.

[47] Laxmikant Kalé and Sanjeev Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++”. In: *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. New York, NY, 1993.

[48] Amir Kamil. *A Team Analysis Proposal for Recursive Single Program, Multiple Data Programs*. Tech. rep. UCB/EECS-2012-183. University of California, Berkeley, 2012.

[49] Amir Kamil. “Analysis of Partitioned Global Address Space Programs”. MA thesis. University of California, Berkeley, 2006.

[50] Amir Kamil. *The Hierarchical SPMD Programming Model*. Tech. rep. UCB/EECS-2011-28. University of California, Berkeley, 2011.

[51] Amir Kamil, Jimmy Su, and Katherine Yelick. “Making Sequential Consistency Practical in Titanium”. In: *Supercomputing 2005*. Seattle, WA, 2005.

[52] Amir Kamil and Katherine Yelick. “Concurrency Analysis for Parallel Programs With Textually Aligned Barriers”. In: *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*. Hawthorne, NY, 2005.

[53] Amir Kamil and Katherine Yelick. “Enforcing Textual Alignment of Collectives Using Dynamic Checks”. In: *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*. Newark, DE, 2009.

[54] Amir Kamil and Katherine Yelick. “Hierarchical Pointer Analysis for Distributed Programs”. In: *The 14th International Static Analysis Symposium (SAS 2007, Kongens Lyngby*. 2007.

[55] Shoaib Kamil, Derrick Coetzee, and Armando Fox. “Bringing Parallel Performance to Python with Domain-Specific Selective Embedded Just-in-Time Specialization”. In: *Python for Scientific Computing Conference 2011 (SciPy 2011)*. 2011.

[56] Shoaib Kamil et al. “An Auto-tuning Framework for Parallel Multicore Stencil Computations”. In: *In International Parallel and Distributed Processing Symposium (IPDPS*. 2010.

[57] Shoaib Kamil et al. “Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations”. In: *Proceedings of the 2005 Workshop on Memory System Performance*. MSP ’05. Chicago, Illinois: ACM, 2005, pp. 36–43. ISBN: 1-59593-147-3. DOI: 10.1145/1111583.1111589. URL: http://doi.acm.org/10.1145/1111583.1111589.

[58] Shoaib Kamil et al. “Portable Parallel Performance from Sequential, Productive, Embedded Domain-Specific Languages”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’12. New Orleans, Louisiana, USA: ACM, 2012, pp. 303–304. ISBN: 978-1-4503-1160-1. DOI: `10.1145/2145816.2145865`. URL: `http://doi.acm.org/10.1145/2145816.2145865`.

[59] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392.

[60] Arvind Krishnamurthy and Katherine Yelick. “Analyses and Optimizations for Shared Address Space Programs”. In: 1996.

[61] Leslie Lamport. “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* 28.9 (1979), pp. 690–691.

[62] *Lawrence Berkeley National Laboratory*. Berkeley, CA. URL: `http://www.lbl.gov`.

[63] Jaejin Lee, Samuel Midkiff, and David Padua. “Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs”. In: *Proceedings of 1999 ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*. Atlanta, GA, May 1999.

[64] Jaejin Lee and David Padua. “Hiding Relaxed Memory Consistency with Compilers”. In: *Parallel Architectures and Compilation Techniques*. Barcelona, Spain, 2001.

[65] Ben Liblit and Alexander Aiken. “Type Systems for Distributed Data Structures”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Boston, Massachusetts, 2000.

[66] Ben Liblit, Alexander Aiken, and Katherine Yelick. “Type Systems for Distributed Data Sharing”. In: *International Static Analysis Symposium*. San Diego, California, 2003.

[67] Yuan Lin. “Static Nonconcurrency Analysis of OpenMP Programs”. In: *First International Workshop on OpenMP (IWOMP 2005)*. Eugene, OR, 2005.

[68] Stephen Masticola and Barbara Ryder. “Non-concurrency Analysis”. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*. San Diego, California, 1993.

[69] John Mccalpin and David Wonnacott. *Time Skewing: A Value-Based Approach to Optimizing for Memory Locality*. Tech. rep. DCS-TR-379. Department of Computer Science, Rutgers University, 1999.

[70] David McQueen and Charles Peskin. “A General Method for the Computer Simulation of Biological Systems Interacting with Fluids”. In: *Biological Fluid Dynamics* (1995).

[71] David McQueen and Charles Peskin. “Shared-Memory Parallel Vector Implementation of the Immersed Boundary Method for the Computation of Blood Flow in the Beating Mammalian Heart”. In: *Journal of Supercomputing* 11.3 (1997), pp. 213–236.

[72] Sabrina Merchant. “Analysis of a Contractile Torus Simulation in Titanium”. MA thesis. University of California, Berkeley, 2003.

[73] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, version 1.1*. 1995. URL: http://www.mpi-forum.org/docs/.

[74] Seung-Jai Min, Costin Iancu, and Katherine Yelick. “Hierarchical Work Stealing on Manycore Clusters”. In: *Fifth Conference on Partitioned Global Address Space Programming Models*. Galveston Island, Texas, 2011.

[75] *National Energy Research Scientific Computing Center (NERSC)*. Berkeley, CA. URL: http://www.nersc.gov.

[76] Robert H. B. Netzer and Barton P. Miller. “What are Race Conditions?: Some Issues and Formalizations”. In: *ACM Lett. Program. Lang. Syst.* 1.1 (1992), pp. 74–88. ISSN: 1057-4514. DOI: http://doi.acm.org/10.1145/130616.130623.

[77] Robert Numrich and John Reid. *Co-Array Fortran for Parallel Programming*. Tech. rep. RAL-TR-1998-060. Rutherford Appleton Laboratory, 1998.

[78] Michael O’Boyle. and Elena Stöhr. “Compile Time Barrier Synchronization Minimization”. In: *Parallel and Distributed Systems, IEEE Transactions on* 13.6 (2002), pp. 529–543.

[79] Robert O’Callahan and Jong-Deok Choi. “Hybrid Dynamic Data Race Detection”. In: *PPoPP ’03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego, California, USA: ACM Press, 2003, pp. 167–178. ISBN: 1-58113-588-2. DOI: http://doi.acm.org/10.1145/781498.781528.

[80] *OpenMP Specifications*. URL: http://www.openmp.org.

[81] Heidi Pan. “Cooperative Hierarchical Resource Management for Efficient Composition of Parallel Software”. PhD thesis. Massachusetts Institute of Technology, 2010.

[82] Heidi Pan, Benjamin Hindman, and Krste Asanović. “Composing Parallel Software Efficiently with Lithe”. In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, pp. 376–387. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806639. URL: http://doi.acm.org/10.1145/1806596.1806639.

[83] Heidi Pan, Benjamin Hindman, and Krste Asanović. “Lithe: Enabling Efficient Composition of Parallel Libraries”. In: *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*. HotPar’09. Berkeley, California: USENIX Association, 2009, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855591.1855602.

[84] *Portable Hardware Locality (hwloc)*. URL: http://www.open-mpi.org/projects/hwloc/.

[85] Christoph von Praun and Thomas R. Gross. “Static Conflict Analysis for Multi-threaded Object-Oriented Programs”. In: *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. San Diego, California, USA: ACM Press, 2003, pp. 115–128. ISBN: 1-58113-662-5. DOI: http://doi.acm.org/10.1145/781131.781145.

[86] Thomas Reps. “Program Analysis via Graph Reachability”. In: *ILPS ’97: Proceedings of the 1997 International Symposium on Logic Programming*. Port Washington, New York, United States: MIT Press, 1997, pp. 5–19. ISBN: 0-262-63180-6.

[87] Radu Rugina and Martin Rinard. “Pointer Analysis for Multithreaded Programs”. In: *PLDI ’99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. Atlanta, Georgia, United States: ACM Press, 1999, pp. 77–90. ISBN: 1-58113-094-5. DOI: http://doi.acm.org/10.1145/301618.301645.

[88] Vijay Saraswat. *Report on the Experimental Language X10, Version 0.41*. IBM Research. Feb. 7.

[89] John E. Savage and Mohammad Zubair. “A Unified Model for Multicore Architectures”. In: *Proceedings of the 1st international Forum on Next-Generation Multicore/Manycore Technologies*. IFMT ’08. Cairo, Egypt: ACM, 2008, 9:1–9:12. ISBN: 978-1-60558-407-2. DOI: 10.1145/1463768.1463780. URL: http://doi.acm.org/10.1145/1463768.1463780.

[90] Stefan Savage et al. “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”. In: *ACM Trans. Comput. Syst.* 15.4 (1997), pp. 391–411. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/265924.265927.

[91] Dennis Shasha and Marc Snir. “Efficient and Correct Execution of Parallel Programs that Share Memory”. In: *ACM Trans. Program. Lang. Syst.* 10.2 (1988), pp. 282–312. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/42190.42277.

[92] Stephen F. Siegel and George S. Avrunin. “Modeling Wildcard-Free MPI programs for Verification”. In: *PPoPP ’05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Chicago, IL, USA: ACM, 2005, pp. 95–106. ISBN: 1-59593-080-9. DOI: http://doi.acm.org/10.1145/1065944.1065957.

[93] *StencilProbe: A Microbenchmark for Stencil Applications*. URL: http://www.cs.berkeley.edu/~skamil/projects/stencilprobe/.

[94] Jimmy Su. “Optimizing Irregular Data Accesses for Cluster and Multicore Architectures”. PhD thesis. University of California, Berkeley, 2010.

[95] Zehra Sura et al. “Automatic Implementation of Programming Language Consistency Models”. In: *Proceedings of the 15th Workshop on Workshop on Languages and Compilers for Parallel Computing*. College Park, Maryland, 2002.

[96] Zehra Sura et al. "Compiler Techniques for High Performance Sequentially Consistent Java Programs". In: *Proceedings of the 2005 ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*. Chicago, Illinois, 2005.

[97] Yuan Tang et al. "The Pochoir Stencil Compiler". In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 117–128. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989508. URL: http://doi.acm.org/10.1145/1989493.1989508.

[98] Pilar de la Torre and Clyde P. Kruskal. "Submachine Locality in the Bulk Synchronous Setting (Extended Abstract)". In: *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*. Euro-Par '96. London, UK, UK: Springer-Verlag, 1996, pp. 352–358. ISBN: 3-540-61627-6. URL: http://dl.acm.org/citation.cfm?id=646669.701085.

[99] Chau-Wen Tseng. "Compiler Optimizations for Eliminating Barrier Synchronization". In: *SIGPLAN Not.* 30.8 (1995), pp. 144–155. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/209937.209952.

[100] United States Department of Defense. *Reference Manual for the Ada Programming Language*. Tech. rep. ANSI/MIL-STD-1815A. Washington, D.C., 1983.

[101] Leslie G. Valiant. "A Bridging Model for Multi-core Computing". In: *J. Comput. Syst. Sci.* 77.1 (Jan. 2011), pp. 154–166. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2010.06.012. URL: http://dx.doi.org/10.1016/j.jcss.2010.06.012.

[102] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. "Algorithms for Parallel Memory II: Hierarchical Multilevel Memories". In: *ALGORITHMICA* 12 (1993), pp. 148–169.

[103] Tong Wen and Phillip Colella. "Adaptive Mesh Refinement in Titanium". In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*. 2005.

[104] John Whaley and Monica Lam. "Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. Washington, DC, USA, 2004.

[105] David Wonnacott. "Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations". In: *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*. IPDPS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 171–. ISBN: 0-7695-0574-0. URL: http://dl.acm.org/citation.cfm?id=846234.849346.

[106] Yonghong Yan et al. "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement". In: *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*. Newark, DE, 2009.

[107] Siu Man Yau. "Experience in Using Titanium for Simulation of Immersed Boundary Biological Systems". MA thesis. University of California, Berkeley, 2002.

[108] Katherine Yelick et al. “Parallel Languages and Compilers: Perspective from the Titanium Experience”. In: *The International Journal of High Performance Computing Applications* 21.2 (2007).

[109] Katherine Yelick et al. “Productivity and Performance Using Partitioned Global Address Space Languages”. In: *Parallel Symbolic Computation 2007*. London, Ontario, 2007.

[110] Katherine Yelick et al. “Titanium: A High-Performance Java Dialect”. In: *Workshop on Java for High-Performance Network Computing*. Stanford, California, 1998.

[111] Yuan Zhang and Evelyn Duesterwald. “Barrier Matching for Programs with Textually Unaligned Barriers”. In: *PPoPP ’07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Jose, California, USA: ACM, 2007, pp. 194–204. ISBN: 978-1-59593-602-8. DOI: http://doi.acm.org/10.1145/1229428.1229472.

[112] Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao. “Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers”. In: *Languages and Compilers for Parallel Computing, 20th International Workshop, LCPC 2007*. Vol. 5234. Lecture Notes in Computer Science. 2008, pp. 95–109. ISBN: 978-3-540-85260-5.

[113] Gengbin Zheng et al. “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers”. In: *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*. Washington, DC, 2010.

[114] Yingchun Zhu and Laurie J. Hendren. “Communication Optimizations for Parallel C Programs”. In: *PLDI ’98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. Montreal, Quebec, Canada: ACM Press, 1998, pp. 199–211. ISBN: 0-89791-987-4. DOI: http://doi.acm.org/10.1145/277650.277723.

وَآخِرُ دَعْوَانَا بِتَوْفِيقِ رَبِّنَا   أَنِ الحَمْدُ لِلهِ الَّذِي وَحْدَهُ عَلَا

وَبَعْدُ صَلَاةُ اللهِ ثُمَّ سَلَامُهُ   عَلَى سَيِّدِ الخَلْقِ الرِّضَا مُتَنَخِّلَا

مُحَمَّدٍ المُخْتَارِ لِلْمَجْدِ كَعْبَةً   صَلَاةً تُبَارِي الرِّيحَ مِسْكًا وَمَنْدَلَا

وَتُبْدِي عَلَى أَصْحَابِهِ نَفَحَاتِهَا   بِغَيْرِ تَنَاهٍ زَرْنَبًا وَقَرَنْفُلَا