University of Arkansas, Fayetteville
ScholarWorks@UARK

Computer Science and Computer Engineering Undergraduate Honors Theses | Computer Science and Computer Engineering

5-2008

Holistic Characterization of Parallel Programming Models in a Distributed Memory Environment

Christopher Bryan
University of Arkansas, Fayetteville

Follow this and additional works at: http://scholarworks.uark.edu/csceuht

Part of the Programming Languages and Compilers Commons

Recommended Citation
Bryan, Christopher, "Holistic Characterization of Parallel Programming Models in a Distributed Memory Environment" (2008). *Computer Science and Computer Engineering Undergraduate Honors Theses*. 15.
http://scholarworks.uark.edu/csceuht/15

This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu.

Holistic Characterization of Parallel Programming Models in a Distributed Memory Environment

Chris Bryan

cjb04@uark.edu

April 23, 2008

An honor's thesis submitted in partial fulfillment

of the requirements for the degree of Bachelor of

Science in Computer Science

By

Chris Bryan

May 2008,

University of Arkansas

Fayetteville, Arkansas

Contents

1 Introduction **2**

1.1 Recent Emphases in Productivity, Leading to Holistic Characterization 2

1.2 Holistic Characteristics Themselves 3

1.3 Impact 4

2 Background **6**

2.1 Parallel Computing Overview 6

2.1.1 Data and Task Parallelism 7

2.1.2 Distributed Memory and Message Passing 9

2.1.3 Shared Memory 10

2.1.4 Distributed-Shared Memory 11

2.1.5 Today's HPC landscape 12

2.2 MPI Overview 15

2.3 Titanium Overview 16

2.4 Fortress Overview 19

3 Methodology **22**

3.1 Performance Metrics 22

3.2 Programmability Metrics 23

3.2.1 Lines of Code and Number of Characters used 24

3.2.2 Sequential-to-Parallel Conversion effort 24

3.2.3 Parallel Conceptual Complexity 25

3.2.4 Code Development Time 28

3.2.5 Other Metrics Not Implemented Here 28

3.3 Coding Style and Expressivity Standards 29

3.4 Kernel Specifications 30

3.4.1 Matrix Multiply 30

3.4.2 Matrix Transform 34
3.5 Closing Methodology Notes 37
4 Results and Analysis **38**
4.1 Hardware Setup and Testing Setup 38
4.2 Benchmark Results and Analysis 40
4.2.1 Shared Memory Multiply Results 40
4.2.2 Distributed Memory Multiply Results 45
4.2.3 Tusk/Tusk-Sun Multiply Results 48
4.2.4 Shared Memory Transform Results 51
4.2.5 Distributed Memory Transform Results 54
4.2.6 Tusk/Tusk-Sun Transform Results 58
4.2.7 Performance Results Notes 59
4.3 Productivity Results 60
4.3.1 Lines of Code and Number of Characters 61
4.3.2 Sequential-to-Parallel Conversion Effort 62
4.3.3 Parallel Conceptual Complexity 63
4.4 Code Development Time 66
4.4.1 Implementations Problems 66
4.4.2 Productivity Notes 67
4.5 Holistic Evaluation 68
5 Conclusions and Future Work **69**
6 Appendix A : Program Code **71**
6.1 Fortress Code for Matrix Multiplication 71
6.2 Fortress Code for Matrix Transform 72
6.3 Sequential Java Code for Matrix Multiplication 72
6.4 Sequential Java Code for Matrix Transform 73

6.5 Sequential C Code for Matrix Multiply 73
6.6 Sequential C Code for Matrix Transform 75
6.7 "Naive" Titanium Code (Ti-Naive) for Matrix Multiply 76
6.8 "Real" Titanium Code (Ti-Real) for Matrix Multiplication 77
6.9 Titanium Code for Matrix Transform 78
6.10 MPI Code for Matrix Multiplication 79
6.11 MPI Code for Matrix Transform 80

7 Appendix B : Runtime Results 82

7.1 Multiplication Runtimes 83
7.1.1 SM MPI 83
7.1.2 SM Ti-Real 83
7.1.3 SM Ti-Naive 83
7.1.4 SM Ti-Real 84
7.1.5 DM MPI 84
7.1.6 DM Ti-Real 84
7.1.7 DM Ti-Naive 84
7.1.8 TS-MPI 85
7.1.9 TS-Ti-Real 85
7.2 Transform Runtimes 85
7.2.1 SM MPI 85
7.2.2 SM Titanium 85
7.2.3 DM MPI 86
7.2.4 DM Titanium 86
7.2.5 TS-MPI 86
7.2.6 TS-Titanium 86

List of Figures

1 PGAS memory model . . . 13
2 PGAS execution model . . . 14
3 Matrix Multiplication : A * B = C . . . 31
4 Matrix Transform . . . 35
5 SM Multiply of length 128x128 . . . 41
6 SM Multiply of length 256x256 . . . 41
7 SM Multiply of length 515x512 . . . 42
8 SM Multiply of length 1024x1024 . . . 42
9 SM Multiply of length 2048x2048 . . . 44
10 DM Multiply of length 128x128 w Naive Titanium Code . . . 45
11 DM Multiply of length 128x128 . . . 47
12 DM Multiply of length 256x256 . . . 47
13 DM Multiply of length 512x512 . . . 47
14 DM Multiply of length 1024x1024 . . . 47
15 Tusk/Tusk-Sun Comparison Runs - Multiply with length 256 . . . 49
16 Tusk/Tusk-Sun Comparison Runs - Multiply Matrix 1024 . . . 49
17 SM Transform - 128x128 Matrix . . . 52
18 SM Transform - 256x256 Matrix . . . 52
19 SM Transform - 512x512 Matrix . . . 53
20 SM Transform - 1024x1024 Matrix . . . 53
21 SM Transform - 2048x2048 Matrix . . . 53
22 DM Transform - 128x128 Matrix . . . 55
23 DM Transform - 256x256 Matrix . . . 55
24 DM Transform - 512x512 Matrix . . . 56
25 DM Transform - 1024x1024 Matrix . . . 56
26 DM Transform - 2048x2048 Matrix . . . 57

27 Tusk/Tusk-Sun Comparison Runs - Transform of length 256x256 58
28 Tusk/Tusk-Sun Comparison Runs - Transform of length 1024x1024 58
29 LoC Multiply 61
30 NoC Multiply 61
31 CpL Multiply 61
32 LoC Transform 62
33 NoC Transform 62
34 CpL Transform 62

List of Tables

1 MPI Example Code Complexity Score 27

2 Percent Conversion Efficiencies 62

3 Titanium Naive Matrix Multiply Parallel Complexity 63

4 Titanium Real Matrix Multiply Parallel Complexity 64

5 Fortress Matrix Multiply Parallel Complexity 64

6 MPI Matrix Multiply Parallel Complexity 64

7 Titanium Matrix Transform Parallel Complexity 65

8 Fortress Matrix Transform Parallel Complexity 65

9 MPI Matrix Transform Parallel Complexity 66

Acknowledgements

This project was supported in part by faculty research equipment awards from Sun Microsystems, Dell Corporation, and an undergraduate research award from the Honors College of the University of Arkansas, Fayetteville. In addition to this, this work owes enormous thanks to Wesley Emeneker and Dr. Amy Apon, who provided invaluable assistance, guidance, and patience along the way.

Abstract

The popularity of cluster computing has increased focus on usability, especially in the area of programmability. Languages and libraries that require explicit message passing have been the standard. New languages, designed for cluster computing, are coming to the forefront as a way to simplify parallel programming. Titanium and Fortress are examples of this new class of programming paradigms. This work holistically characterizes these languages and contrasts them with the standard model of parallel programming, and presents benchmark results of small computational kernels written in these languages and models.

1 Introduction

High Performance Computing (HPC) with clusters of commodity computers has experienced enormous growth in recent years in scientific and business computing environments. Despite this growth, little work has been done in simplifying usability, and HPC is still difficult. Writing programs for serial execution can be hard. Difficulties are only compounded when writing correct parallel programs. Each node in an HPC cluster is independent from every other node. In order to process data in parallel, the nodes have to share data. Sharing is usually done with message passing, a technique where data is sent between nodes over a network connecting them. The standard parallel programming model has been one of explicit message passing, the widely used Message Passing Interface (MPI) being the most popular in HPC. As HPC's popularity increases, and the need for parallel programming increases (especially with multicore architectures emerging), enhancing parallel usability becomes increasingly important [1, 2].

This work studies programmer productivity and language usability for the standard MPI and two new developing programming languages, Titanium[3] and Fortress [4], that are being developed to simplify the process of parallel programming. These languages have stated goals of enhancing programmer productivity, and *programmability*, not only in their ability to do HPC, but also by having an emphasis on high usability. This work examines these three parallel programming models in a holistic way, studying and analyzing both language usability and benchmark performance for two computational algorithms, a naive matrix multiply and a communication-intensive matrix transformation.

1.1 Recent Emphases in Productivity, Leading to Holistic Characterization

Many historical analyses of parallel models consist of optimized benchmark codes and resulting runtimes. With the rise in popularity of parallel programming on HPC systems, it has become clear that runtime is no longer the only metric that counts. Programmer productivity should also be considered. The Defense Advanced Research Projects Agency (DARPA) High Productivity Computer

Systems (HPCS) initiative has recognized the importance of usability and defines defines productivity as “a combination of programmability, portability and robustness,” [2]. The HPCS initiative has solicited work to develop languages that focus not only on improving program runtime, but development time as well [5, 2]. The two languages studied in this work, Fortress and Titanium, are designed to satisfy these goals.

Titanium (developed at the University of California-Berkeley) has three stated main goals: performance, safety, and expressiveness [3]. Titanium is a parallel version of Java built in the so-termed Partitioned Global Address Space (PGAS) paradigm. This means that data is partitioned across processors, and may be declared as global or local. Fortress, a DARPA HPCS solicited language, is stated as being “designed for producing robust high-performance software with high programmability,” [4]. Fortress is an entirely new language built upon the concept of mathematical notation programming, and incorporates parallelism as an implicit part of the language that may also be explicitly exploited. Both of these languages exemplify the new wave of parallel programming that focuses on usability without neglecting performance.

The HPCS program’s goals are runtime and usability. Execution time is easy to measure, productivity and programmability are not. These characteristics are much more qualitative in nature. Measurement of the these characteristics, or the *expressivitiy* of a language, is vague and often a loosely defined term. Research in this topic has not produced a widely-accepted productivity standard for measurement, or drawn any firm conclusion about what makes one language more or less expressive than another, even in sequential languages [6, 7, 8, 9]. As a result, the general HPC community has preferred to quantitative benchmarks such as the NAS Parallel Benchmarks (NPB) [10], Linpack [11] and High-Performance Linpack (HPL) [12], and the High Performance Computing Challenge (HPCC) [13] that provide objective operations-per-second and timing results.

1.2 Holistic Characteristics Themselves

Much work into HPC system performance has been done at the expense of usability research. This work looks at programmability as well as performance. Much of this work will examine the

holistic, or all-inclusive, qualities of Titanium, Fortress, and MPI. This is done through the use of program chrestomathy, which is the development of similar programs written in the different languages for the purpose of demonstrating differences in syntax, semantics, parallel conceptualization, idioms, and performance.

This work shows that specifying a set of productivity and performance metrics allows for a holistic comparison of languages. To do this, we take a set of generalized, loosely applicable characterizations:

- lines of code
- characters in code
- parallel constructors used
- documentation
- sequential-to-parallel complexity

and then implement a set of simple benchmark kernels. The process of writing the kernel and measuring these characteristics leads to a conceptualization of the language's programmability, as well as a general idea of the complexity of programming required to implement a certain application.

The focus of this work is on overall usability. However, application performance will not be overlooked. Any parallel programming model or language that wishes to be taken seriously for scientific computing must be able to supplement its development productivity with computational results. Therefore, the kernels implemented, a matrix multiply and matrix transformation, will be tested for both programmability and performance.

1.3 Impact

The emphasis in this work is on developing real conclusions from holistic productivity measurements. However, the sum of these characterizations cannot be accumulated to form some sort of objective productivity score to say that one parallel language is better than another. Although the

results of this work are qualitative, this research is still valid. It gives a real world sense of what it is like to do parallel programming in these models. In addition, the metrics implemented here may be further used for continuing research on these languages, or they may be rewritten in other parallel programming models for further usability or performance testing.

2 Background

Modern high performance computing is defined by the use of parallelism. Parallel computing is a form of computing in which multiple processors work simultaneously on a single problem or application [14]. Parallel computation can be carried out by a single computer with multiple internal processors, or multiple computers that communicate over a network. MPI, Titanium, and Fortress all use parallelism in different ways to execute programs. This chapter gives an overview and history of parallel computing, and defines the types of parallel computation. The history and motivation for MPI, Titanium, and Fortress are discussed, and are examined in the context of the current state of HPC langauge development.

2.1 Parallel Computing Overview

Parallel computing is not a new idea. The topic has been discussed since the late 1950s [15, 16], and the idea of a parallel computer was described in the early 1960s [17], with much more work and important progress happening as that decade went on [18, 19, 20].

Historically, parallel processing has either been done on symmetric multi-processor (SMP) machines with large number of processors sharing a memory, or on vector machines. This approach works well for many tasks, but is prohibitively expensive and has limited scalability. Distributed systems made of physically independent processors and memories evidence better scalability and price/performance as long as the programmer is sufficiently clever.

Distributed systems make use of massively parallel processors (MPPs) all operating together. This provides more cost-efficient scalability than the monolithic "big iron" machines of the 1980s [21]. They do this by sending messages over a network to share data, a technique commonly called message passing.

The migration from large SMP and vector machines began in the 1980s and culminated with Beowulf clusters, a type of system built using commercial-off-the-shelf components striving to achieve HPC at a low cost [22, 21]. Beowulf clusters (simply denoted as clusters for the rest of

this paper) increased the emphasis on high-performance message passing.

Explicit message passing is generally regarded as hard to use by programmers. Data must be explicitly decomposed and passed between memories. This requires fine-grained control over a program, and can become much more complicated than simply having a single address space that is accessible by all processors, as is done with shared memory. Therefore, hybrids have been developed and researched, including distributed-shared memory, which simulates shared memory on a distributed system. In all of this, there are two fundamental types of parallelism that message passing can implement, *data parallelism*, and *task parallelism*.

2.1.1 Data and Task Parallelism

Data and task parallelism are two fundamental approaches to parallel programming. Their difference is in the way they approach the parallelism. Data parallelism uses multiple processors to perform a set of computations on different data sets. Task parallelism uses multiple processors to implement different paths of computation, i.e. tasks, in parallel.

In data parallelism, a single set of instructions is executed in parallel by different processors using distributed data. The parallelism comes entirely from data partitioning. Data parallelism generally scales well to larger problems [14]. Data parallel applications also resemble a single program, which enhances *programmability*. The data partitioning can be handled by the compiler instead of the programmer, and so the program appears more like a sequential program in the way that it is programmed.

Flynn's taxonomy, a classification system based on execution type, defines this as Single Instruction, Multiple Data [20]. Data parallelism can be implemented as *loop-level parallelism*. In this model, the programmer is not responsible for communication between the processors, only data distribution. Some notable implementations of data parallelism include High Performance Fortran [23], ZPL [24], NESL [25], HPJava [26], and forall loops in OpenMP [27].

Data parallelism is attractive for its semantic simplicity. Hillis and Steele Jr. discuss several data parallel algorithms [28]. There are, however, factors that limit the success of data parallel

languages.

1. The number of algorithms that may be performed is limited.

2. By being limited to performing identical operations in parallel, computations like divide-and-conquer and adaptive algorithms are challenging to implement.

3. Data parallel languages rely on sophisticated compiler and runtime support that take control away from programmers [29, 30, 31].

In contrast to data parallelism, task parallelism is defined by a program having multiple paths of parallel execution. Each process, however, is free to follow its own path of execution. This is known as Multiple Instruction, Multiple Data in Flynn's taxonomy [20, 14]

In task parallelism, a program will spawn *processes* of execution that execute code in parallel using multiple processors. The processes may all execute the same instructions, or may execute different ones. Parallel programming usually maps these processes (or *threads*, as they are sometimes called in shared memory), in an associative relationship with processors. This means that the process is paired with a processor (usually in a one-to-one mapping), and the processes/processors all work in parallel.

Some forms of task parallelism allow dynamic process creation, such as pthreads[32], Java threads [33], MPI-2 [34], Charm++ [35], CC++ [36], Fortress [4], and OpenMP's parallel blocks [27]. When processes are static, that is, there is a fixed number of processes throughout the program, the Single Program, Multiple Data (SPMD) model results [14]. With SPMD, all processes are created at program startup and execute the same program, perhaps branching on conditional statements to execute different code. Processes may be synchronized through use of barriers or communication calls, but otherwise continue their own paths of execution. Examples of SPMD programming models include MPI [37], SHMEM [38], and Titanium [39].

Data and task parallelism may be implemented where the view of memory is shared or distributed. A single view of memory by processes gives a global address space (GAS) addressable by all processes. Distributed memory requires processes use message passing to communicate with

each other. Message passing is usually used on distributed memory systems, and GAS programming is usually implemented in shared memory systems. When GAS is implemented on distributed memory systems (by use of implicit message passing), Distributed-Shared Memory (DSM) is the result.

2.1.2 Distributed Memory and Message Passing

Distributed memory machines are made of independent nodes that have physically distinct components, such as memories, CPUs, and disks. They must share data by message passing over the network.

Distributed memory computer clusters have become the dominant HPC architecture in recent years, especially as commodity cluster research has gained focus [40, 41, 42]. Commodity clusters are defined by having their computer nodes and interconnects having commercial-off-the-shelf (COTS) components [43]. Commodity clustering is a more cost-efficient way to get high-end computation [43, 21]. Two important programs in the development of commodity clusters were the Berkeley NOW (Network of Workstations) [41] and work by Thomas Sterling and Donald Becker, who coined the term Beowulf clusters [22].

Distributed memory parallel programming usually utilizes the message passing model for program communication and coordination. In message passing, the programmer is left to explicitly divide data and work across processes (which are mapped to processors), and is required to manage communication between them [14].

Parallel Virtual Machine (PVM) was an early notable implementation of message passing [44]. PVM's focus was on having the abilitity to communicate between a loosely-coupled, heterogenous network of workstations, to achieve parallelism. Its emphasis was on providing a distributed computing environment. The MPI standard was introduced in 1994, and quickly became the *de facto* standard for HPC message passing [45]. MPI was specifically designed with HPC in mind, and therefore it superceded PVM in this realm. Gropp and Lusk provide a good review on goals, differences, and similarities between MPI and PVM [46].

The message passing model has traits that make it attractive for parallel computation. Any type of parallelizable computation may be rewritten using an abstract model of send and receieve calls between processors, and message passing implementations can execute on both shared and distributed memory machines [47, 48]. Additionally, message passing implementations like MPI and PVM have high *portability* since they are implemented across a variety of hardware platforms [48, 44]. However, because of its explicitness, message passing requires fine grained control over data and program flow. This is bad because it is required. If only allowed, it could be good, but instead programmers are forced to write restricted code that adheres to communication limitations.

2.1.3 Shared Memory

In the shared-memory paradigm, there is one addressable storage space to which processors have access. Data must be kept consistent from simultaneous manipulation by the use of some form of locking, and communication usually happens through the use of loads and stores between processors and memory.

The *global view* of data that shared memory provides means that all processors see and have access to the same data. From a programmer's view, it is very convenient then to share data. However, it is difficult and expensive to scale "true" shared-memory machines to more than a few tens of processors while still having memory access time be uniform [45, 14].

There are three notable methods for programming with shared memory systems. They are Unix heavyweight processes, threads, and OpenMP. Unix processes were one of the earliest ways to achieve task parallelism [17]. In Unix processes, there are two calls that are important, **fork** and **join**. A fork statement generates a new path of parallel execution by the Unix system by creating a child process. A join statement would terminate the child. The fork call creates an exact copy of the parent process, including variables, except for a unique process ID. Processes share data through explicit means that are written by the programmer, and also have their own private memories.

Threads can be thought of as lightweight processes working within a single process. Threads are much less memory intensive than processes [49, 50]. And, because threads share the same

memory space, they have a global view of variables, a GAS view of programming [14]. While explicit communication is not needed between threads, race conditions for data access and modification are possible. It is the programmer's responsibility to ensure that this does not happen. Many threads implementations, like Posix pthreads [32], contain ways to manage threads and handle synchronization issues by using mutexes, locks, and condition variables. Though threading is better than Unix processes, the threading standard (with pthreads) is not suitable for the scientific community. Pthreads has no Fortran bindings, it is too low-level, it doesn't support data parallelism, and although much faster than Unix processes, is still not performance oriented [27]. Therefore, the OpenMP standard was introduced.

OpenMP is a higher level Application Programmer Interface (API) standard. It defines a set of directives that the compiler transforms into parallel code. OpenMP is built on top of threads and provides a higher-level way to do parallel processing on a shared memory machine. In OpenMP, the parallelization of a program is done with **pragma** directives, which are set at the start of blocks of parallelized code. OpenMP makes it easy to parallelize existing C/C++ and Fortran code, and has support for data parallelism through parallel **for** loops [27, 51], as well as dynamic task parallelism through **pragma** defined blocks. OpenMP is the most widely used shared memory model for parallel HPC today [45, 52].

2.1.4 Distributed-Shared Memory

The hybrid of shared and distributed memory is the distributed-shared memory (DSM) model. In DSM, there may be physically distributed and separate memories, but from a programming standpoint there is a global view of memory shared between processors [14]. With a global address space, programming may be done like the system is composed of shared memory, and message passing is not explicit. The challenge of DSM implementations is providing a level of abstraction to shared memory that still produces efficiently running code. Although it can make use of compiler optimizations, one-way communications, and remote-direct memory accesses to help speed up computation, DSM has several drawbacks including communication overhead, network latency,

false sharing, coherence and page faults [53, 14].

DSM has been implemented in many different ways. Sometimes the entire system image is virtualized [54, 55]. Software languages and libraries [56, 57, 58] provide another option. Hardware [59, 60] is a third option, but none of these has been widely successful in widespread implementation.

DSM presents a view of multiple machines as if they were one. Historically, DSM has been a niche market in HPC. This is because extracting adequate performance on DSM systems is usually extremely difficult [14]. DSM has an "undisciplined" view of data in that any variable may be accessed at any time by any processor [1]. These problems negatively impact performance to a degree that makes DSM basically unusable for HPC.

2.1.5 Today's HPC landscape

Currently, MPI is the dominant model for distributed programming and OpenMP is the most popular shared memory parallel model [1, 52]. Shared memory programming is generally considered to be easier than message passing. However, shared memory machines are very expensive [61]. Commodity clustering can lead to more computing power at lower costs, but requires message passing. Therefore, harder work is required to do message passing, but it is done anyway because it is the best way currently to get high performance at a reasonable cost [62]. It would be nice if ease of programming in the shared memory paradigm could be used to give high performance applications on distributed memory. Ideally, we want a language that has high programmability, like GAS programming, but also high performance like MPI.

There have been numerous approaches to producing these high productivity languages. Shared address space and DSM are one. However, pure DSM has many problems and has been discarded as a serious HPC option. Therefore, research is trying to take the best parts of GAS and develop models that are hybridized with message passing, forming models that have message passing performance with the ease of shared memory programming. These models will have the convenience of implicit parallelism through a global view of data with implicit communication, but achieve

high performance and scalability through use of clever compiler optimization and programmer control of data layout and intraprocessor communication. The DARPA HPCS initiative and the development of Partitioned Global Address Space languages like Titanium and Unified Parallel C are results of this research.

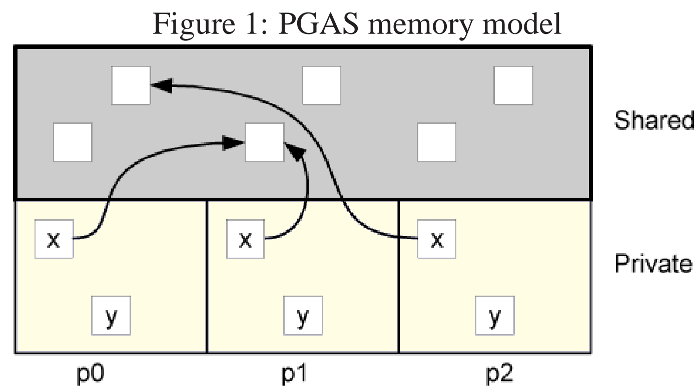
In 2002, DARPA launched the HPCS program. Its primary goal is to develop systems that improve the overall productivity of high performance computing by reducing code development time and complexity while retaining good performance.

The program itself is broken into three phases. Phase II of the program, which lasted through July 2006, had funding for three languages being developed by three different vendors- Chapel from Cray [63], X10 from IBM, and Fortress from Sun. Fortress was dropped from the program at the start of Phase III, and is now an open source project under the direction of Sun [64].

The importance of the DARPA project is that it recognizes the need for language productivity. The three main HPCS languages all feature GAS programming, while allowing the ability to perform data and task parallelism, through loops or dyanmic process creation.

A specialized approach to DSM is the partitioned global address space (PGAS). In PGAS implementations, there is a shared memory space that all processes may use to share variables, like in shared memory, and at the same time facilitate implicit communication, like in DSM. Additionally, there is also a private memory region for each process.

Figure 1: PGAS memory model

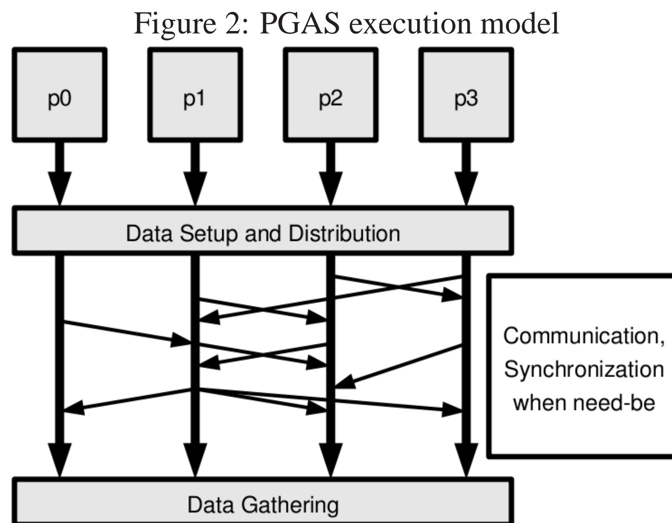


The word “partitioned” indicates that global data variables are divided up and stored in differ-

ent individual process memories. The data has a so-called *affinity* for a particular local memory partition. Processors have fast access to local variables, and slower access time to global variables that may reside in a remote partition of memory. The programmer and/or compiler may control and exploit this data layout to produce optimized communication [65, 66]. Although optimizing communication is not required for correctness, it is critical to achieving good performance on distributed memory systems [67]. A naively-written program may ignore data-locality exploitation, but moving from shared memory to distributed memory would significantly impact performance. In shared memory, all memory is considered local and all references behave as local references [31]. On distributed memory remote puts and gets takes considerable time, even if the data is still physically local to a node, because of global checking overhead associated with all global pointers.

Titanium, Co-Array Fortran [68], and Unified Parallel C [69] are three examples of PGAS languages. Within these languages, programmers use shared variables and may create shared objects as well. One-sided communication is implicitly used to access remote data. This has been shown to lead to faster communication than explicit two-sided message passing, and works well for problems that have data-dependent communications patterns that may be irregular [70, 66]. Titanium, UPC, and CAF all use GASNet for their distributed messaging, which is a low-level network layer that uses one-sided communication [71].

Figure 2: PGAS execution model



Each PGAS language also offers explicit communication functions to broadcast and collect local data between processors as well as barriers to synchronize task execution. A typical program from one of these languages might be structrued as follows (see Figure 2). First, data is scattered to different processes, where each one performs it own threads of execution, using barriers for synchronization. Communication is implicit, using fast one-sided messages when needed. Finally, the data may be collected together to a single process.

2.2 MPI Overview

MPI is an industry standard API specification designed for HPC on multiprocessor and cluster computers. Introduced in 1994 [14], the "primary goal of the MPI specification is to demonstrate that users need not compromise among efficiency, portability, and functionality" [45]. The API designers attempted to collect the best features of earlier message passing systems, improve them where appropriate, and standardize them for the parallel programming community.

History and Development MPI was devloped during 1993 and 1994 by a group of scientists, vendors, and programmers called "the MPI Forum," [72]. What they created was a standard for a message passing library designed for high-performance computation.

A second MPI specification, MPI-2, was defined in 1997 [34]. MPI-2 provides additional features to the MPI-1 specification such as parallel I/O, C++ and Fortran 90 bindings, remote memory access, one-sided communication, and dynamic process management/creation. It should be noted that MPI-2 is not simply MPI version 2. This work does testing only with the MPI-1 specification, and as such the term MPI will refer to the MPI-1 standard.

The MPI standard is not an implementation. That is left to individual vendors, of which there are many. Two notable free implementations are MPICH [73] and LAM/MPI [74]. There are also MPI implementations in many different programming languages [75, 76, 77, 78], and on many different architectures [45, 14]. The entire MPI API (both MPI-1 and MPI-2) contains more than 300 routines [48], although many MPI applications are coded with much smaller subsets of this

[14, 45, 48]. In C/C++ and Fortran, MPI is used as a set of routines that are inserted into source code to control data communication between processes [14].

Parallelism and Communication In a program using MPI, there are a static number of *processes*, which are generally distributed among processors (this is left up to individual implementations, however). Communication happens through send and receive calls, either through point-to-point messages, or collective operations such as *broadcasts*, *reductions*, and *scatters*. This processor-centric method allows for full control of communication and parallelism. Using MPI, a programmer explicitly decomposes the data processing and sharing. Of widely used parallel models for HPC, MPI is in the class of most explicit [79, 80]. Because of this, MPI has been referred to as the "assembly language" for parallel programming [48].

Other Language Notes MPI has been a very successful approach for achieving parallelism in programs. It is widespread, portable, and achieves high performance. It has been described as a "complete" model, whereby any parallel algorithm may be implemented [48]. Yet, MPI has many difficulties. Some common issues raised about MPI include its complexity (as measured by number of functions called), performance costs (especially with regard to communicating small messages) and lack of compile and runtime help [48, 14, 81]. For a programming model to be a successor to MPI, it needs to have the performance, scalability, and completeness of MPI, yet do so in a more elegant and programmable fashion that raises overall productivity.

2.3 Titanium Overview

Titanium is a language designed for "high-performance parallel scientific computing" [3]. Titanium is based on the Java programming language. The main thing that Titanium adds is SPMD parallelism with PGAS support.

History and Development The Titanium project began in 1995 with the objective of building a high-level language on the experience of GAS languages such as Split-C [82], AC [83], and

CC++ [36] [31, 3]. The motivation behind Titanium's design was "to create a language design and implementation enabling portable programming for a wide range of parallel platforms that strikes an appropriate balance between expressiveness, user-provided information about concurrency and memory locality, and compiler and runtime support for parallelism" [31]. The goal was to offer Java's object orientation with strong typing and safe memory management, while allowing for local and global data sharing or passing between processes.

Titanium's foremost goal is performance, followed by safety and expressiveness [3]. Safety is meant in the sense that Titanium's compiler not only detects errors statically, but also that it can detect run-time errors accurately. Expressivity is claimed by use of built-ins like multidimensional array support and **foreach** statements, which make for easier and more readable program development, especially with grid-intensive applications.

Most of Titanium's application work has come from development teams that are closely related to the language compiler or language development effort. Notable Titanium programs include a subset of the NAS Parallel Benchmarks [84], a 2D Poisson equation solver [85], Adaptive Mesh Refinement programs [86, 87], and an Immersed Boundary simulation [88].

Parallelism and Communication The SPMD model of programming means that a Titanium program has a fixed number of *processes* associated with it, in the same way as MPI-1. That is, processes cannot be created dynamically. Within this, the processes themselves are not required to be executing in a tightly synchronized, step-by-step basis. Different processes may follow their own paths of execution, taking loops different amounts of time, just like MPI.

Titanium processes communicate with each other through *global* variables and data structures they share. These processes may transparently read and write data that reside on other proccesors using implicit, one-sided communication. Titanium also has constructs for facilitating explicit collective communcation between processes using *exchange* and *broadcast* calls, and can use *barrier* calls to enforce explicit process synchronization. Additionally, the Titanium keyword *single* is used to ensure that desired variables do not have different values among processes, and that desired

calls are made by all processes.

The local memories associated with Titanium processes are given the name regions [39]. Objects are contained in the region of the process that creates them, and pointers are used to reference them. Global variables and objects may be pointed at by any Titanium process. The declaration *local* means that the variable may only be referenced by the allocating process and it resides in the local partition of memory.

The intent is that in shared (or uniprocessor) implementations, there is only one region, and so global and local pointers are equivalent, and on distributed memory there is one region per processor. Variables in Titanium default to global. This makes porting Java programs easily, but may be inefficient in parallel, as global pointers have overhead associated with them. A naive Titanium application without local and global pointer exploitation might not run as well as a Titanium program that did use data locality to its advantage.

The Titanium compiler performs analysis to optimize parallel code execution. This analysis can do things like converting global pointers into less expensive local pointers when possible, overlaps communication with computation, and preserves the illusion of sequential consistency. On shared memory, accesses to the global memory space translate into conventional load/store instructions, while in distributed memory, the GASNet layer is used to handle messaging [71, 3].

Other Language Notes Java was chosen as the base language (as opposed to C or C++) for Titanium because it is a semantically cleaner and simpler language, which makes it easier to extend [31]. The type-safety of Java allows for better optimization in the compiler, as static information can better be used for program analysis parallelization. It should be noted that the Titanium compiler translates code into C. There is no Java Virtual Machine or Just-In-Time compiler that runs Titanium.

Since Titanium is an extension of Java, most basic Java programs may be run as Titanium programs. Titanium does modify and add to some parts of Java though, for the purposes of making efficient parallel code. Extra language features include immutable classes (similar to **structs** in C),

multidimensional arrays, points and domains, single variables, explicit collective communicators, and local and global references [39].

Multidimensional arrays are very important for HPC, so Titanium has included native support for them. In Java, multi-dimensional arrays are represented as arrays of arrays, and only 1-dimensional arrays are fully supported. In Titanium, arrays are indexed by integer tuples known as *points*, and are built on sets of points, called **domains**. Arrays are built with special rectangular domains called a **RectDomains**. Titanium also has a **foreach** command to execute a block of code on each point in a domain, although this is used to ease readability and compiler checking, and does not produce data parallelism (as each **foreach** call is in only one process's execution path). The purpose of a foreach is for easing boundary work, making for less buggy and easier-to-read code.

2.4 Fortress Overview

Fortress is called a ”novel” language for HPC. It is a built-from-the-ground-up effort by Sun, aiming to facilitate the programming of next-generation parallel systems [2].

Fortress is an object-oriented, statically typed language that is interpreted. It uses a “component” system that is similar to classes in Java, contained within a single “Fortress” or program directory. The current compiler runs on top of a Java Virtual Machine, and only a small core of the language specification is currently working. The program structure of Fortress is meant to reflect scienctific notation, and its syntax, semantics, and parallelism reflect that.

History and Development The name Fortress is meant as a play off of the language Fortran, specifically the thought of a “secure Fortran” [4]. The syntax and semantics of Fortress, though built with elements from many programming languages, is designed to resemble mathematical notation as closely as possible. Fortress was formerly part of the DARPA HPCS initiative, and is now an open-source project overseen by Sun.

Sun's philosophy is for Fortress to be a growable language. The language design strategy is

one that wherever possible, language features are to be provided through libraries, as opposed to compiler hardwiring. This assumes that the implementation technology for Fortress will make use of aggressive runtime performance measurement and optimization.

Parallelism and Communication Fortress is a PGAS language. It presents the user with a global address space much like Titanium, instead of explicit message passing like MPI. Fortress uses threads for task parallelism. Unlike Titanium and MPI-1, Fortress allows dynamic thread creation. Furthermore in Fortress, parallelism is the default. Implicitly created threads are used to do parallel operations, like **for** loops and **do** statements, as well as **tuple** expressions, which are a special block construct that execute expressions in parallel. Fortress includes atomic expressions for controlling parallel interactions, or ensuring variables maintain consistency during modification.

Fortress will eventually allow for user-defined data locality with computation, for performance. Especially for arrays, explicitly distributed data structures allow the automation of this. These data structures are specified in *distributions* to different local memories, where they are divided and mapped among processors. Fortress allows explicit data locality using so-called *distributions* or it may be handled by the compiler. In Fortress, this work is delegated to libraries, and even at that, *distributions* handling libraries are not yet available.

Other Language Notes Fortress is built to resemble mathematical notation, and its syntax reflects that. The goal of Fortress is to translate chalkboard math writing into code that works as seamlessly as possible. The language does things like type inference to achieve better readability, operator overloading, and matching mathematical notation. Semicolons are optional, and Fortress may be typed in ASCII or unicode characters.

For example, instead of using a "*" to denote multiplication, as in *a*b*, juxtaposition may be used to mean the same thing, *a b*.

Fortress will also eventually allow values to be defined in terms of physical units or dimensions, and will provide commonly used ones in Fortress standard libraries. A variable *velocityA* can be defined using the units *Meters* / *Seconds*. Physically defined values or dimensions will be statically

checked by the compiler. For example- seconds cannot be added to meters, and the library-supplied dimension *Acceleration* is equivalent to *Velocity* / *Time* types.

3 Methodology

We have discussed the need for new, more usable and productive parallel languages. These languages must make it easier to read, write, optimize, and debug code. In short, new languages must make the process of going from abstract algorithm and mathematical theory to high-performance application as seamless as possible.

Historically, words like *productivity*, *programmability*, and *usability* have been vaguely defined. Most research into programmer productivity has been only loosely connected, and there is no standardized set of productivity benchmarks [7, 89, 6]. However, this should not imply that productivity, programmability, and usability are unimportant. There is a high focus in enhancing these elements of HPC [90, 1, 79]. Programs like the DARPA HPCS program and the Titanium language out of the University of California-Berkeley seek to develop high-performance languages that ease program development time [3, 5].

This work develops a set of productivity metrics that are used to roughly compare the "usability" of different languages. This approach will not let us say Titanium is 80% usable. It will let us say that Titanium is more usable than MPI, or vice versa. Implementing a program chrestomathy can demonstrate semantic and syntactic differences between languages and APIs. A trivial example of programming chrestomathy is seen in the ACM "Hello, World!" project [91]. This thesis presents two relevant scientific applications for comparison between languages.

In this chapter, the performance and productivity benchmarks are explained. Additionally, the kernels that are implemented are discussed.

3.1 Performance Metrics

The performance of the languages will be shown through runtimes. The time required to perform the parallel distribution and computation is measured for each kernel. For each language, native timing mechanisms are used to perform a timing of computation. For both kernels, the computation of the runtime includes all communication, computation, and data gathering that happened in the

program, excepting matrix creation and population. The specific timing functions used are as follows:

- sequential C - **time()**
- MPI (in C) - **MPI_Wtime()**
- sequential Java - **System.currentTimeMillis()**
- Titanium - a Titanium Timer object
- Fortress - **nanoTime()**

Timing results are noted in the "Results and Analysis" chapter. It should be noted that the since the kernels in C, Java, and Fortress were not timed, the timing mechanisms were merely included so as not to skew the results of the other productivity metrics such as lines of code.

3.2 Programmability Metrics

A holistic characterization of a language cannot be based only on performance. The *programmability*, or ease of use, of each language, will be measured with the following metrics. They are described in detail below.

- Lines of code, number of characters used, and characters per line
- Sequential-to-parallel conversion effort
- Parallel conceptual complexity
- Development time

It should be noted that the ideas and distinctions for the first three metrics are related to a paper by Cantonnet et. al., a loose productivity study between UPC and MPI [92]. The specific details of each metric are given below.

3.2.1 Lines of Code and Number of Characters used

Lines of code (LoC) itself is an imperfect programming measurement, and it has received much criticism. However, it is still a valuable metric to record. A LoC measure can further be supplemented by the total number of characters (NoC) used in the application. A single line may contain one statement that is very long and drawn out. In this case, the NoC will compliment the LoC by allowing us to determine the average number of characters per line (CpL) in each kernel, represented simply by:

$$NoC/LoC = CpL \tag{1}$$

Of course, the sophistication put into each line of code cannot be measured by this, merely the physical output required to type the characters. The ease of programming, or *programmability* of each line cannot be measured as well. These are shortcomings that must be kept in mind using LoC, NoC, and CpL as productivity metrics. However, by comparing LoC, NoC, and CpL between programming models, the code size of each benchmark can be shown.

To implement this measurement for each kernel, every line of code is counted, except for blank lines and comments. Every character is also counted, except for comments, and non-syntactically required spaces and tabs.

3.2.2 Sequential-to-Parallel Conversion effort

This metric attempts to describe the effort required to convert code from a sequential base to parallel. To measure this, we will use two efficiency equations defined as:

$$(LoC_{parallel} - LoC_{sequential})/LoC_{sequential} \tag{2}$$

$$(NoC_{parallel} - NoC_{sequential})/NoC_{sequential} \tag{3}$$

These equations show the difference between the sequential code and parallel code. To perform this metric, the kernels will first be represented in its base language.

- MPI - in sequential C code
- Titanium - in sequential Java code
- Fortress - does not have a base language, so this metric is not applied to it

For MPI and Titanium, the sequentialized code will be modified to be parallel, and the effort equation will be applied to it. A program with a lower number will mean that fewer steps are needed for parallelization. The physical effort needed to parallelize the program will then be less. If the program still achieves good performance, then it can reasonably be concluded that more productivity is attained through ease of conversion while still having high performance.

3.2.3 Parallel Conceptual Complexity

"A conceptually complex programming environment is one in which the original parallel application view is obscured as it compares to the original application view, and thus requires additional work to maintain the correspondence with the original problem" [92]. Special effort is required to transform a sequential version of the code into its corresponding parallel version. A conceptually complex model may introduce many conceptual changes or additives that make the sequential and parallel versions recognizable from each other. Parallel conceptual complexity of a program is defined as the amount that a program must be modified from sequential code to make it parallelized. This may take the form of synchronization calls, domain management calls, loops for different tasks, or handling required to partition work among processes. Parallel conceptual complexity can depend greatly on the underlying programming model.

The score is a positive value based on language constructs needed to parallelize a program. A metric based upon this score shows parallel conceptual complexity. To produce this score, each program written will be viewed. Parallel function calls, along with their parameters and task references are counted, and summed together to produce the score.

There are different elements involved in parallelizing code. Elements for distributing data to processes, elements for determining the amount of work, communicators, synchronization calls, and other required functions are needed. We distinguish between them with the following call types, which are slightly modified from the metrics used by Cantonnet et. al [92]:

- *Work Distributors* (WDs) - define how work is done by processes, or distribute tasks among processes. The calls that distribute work are **if**, **for**, Fortress **tuple**, and Titanium **foreach** statements.

- *Data Distributions* (DDs) - define how data is distributed among processes, and define which processes may access data. Calls to **alloc**, **memset**, textbffree, **local**, **shared**, and **private**, as well as extra created objects or data structures, such as locally allocated arrays are data distributors

- *Communicators* (Comms) - perform explicit communication calls between processes. These may be point-to-point **send** and **receives**, or may be collective calls such as **broadcast** or **scatter** that MPI and Titanium provide, or **copy** operations on arrays that Titanium does.

- *Synchronization and Consistency calls* (SCs) - ensure that all computation reaches a point together, or that a variable is synchronized across all processes. These calls are defined as **barriers**, **single** types in Titanium and MPI, and **atomic** constructs in Fortress.

- *Miscellaneous Operations* (Miscs) include other necessary calls within the program to ensure correct parallelization. Calls to **initialize** or to orient the environment (such as finding the number of processes) or to close down the parallel environment, such as MPI's **finalize** call, as well as required library calls, are considered miscellaneous.

Each construct itself is further broken down into parameters, the function call itself, and task references associated with it. Every parallel construct raises parallel complexity if it uses multiple parameters, or makes references to process tasks or ranks.

- Parameters - the number of parameters within each construct used.

- Function calls - the specific function called (should be one for each call)s
- Rank / Size - references to specific processes, a particular process's rank, or the number of processes in the domain.

Here is some example MPI code:

```
14   MPI_Init(&argc, &argv);
15   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
16   MPI_Comm_size(MPI_COMM_WORLD, &p);
17   if (rank%2==0)
18     MPI_Send(3, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
.
.
.
27   if(rank%2==1)
28       MPI_Recv(&data, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
.
.
.
32   MPI_Barrier(MPI_COMM_WORLD);
33   MPI_Finalize();
```

The parallel conceptual complexity is represented in the following chart, which also gives a parallel complexity score for the above code fragment- 37.

Table 1: MPI Example Code Complexity Score

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	2		13	1	6	22	37
Calls	2		2	1	4	9	
R/S	2		2		2	6	
Notes : Score comes 2 if, 1 MPI_Send, 1 MPI_Recv, 1 MPI_Barrier, 1 MPI_Init, 1 MPI_Comm_world, 1 MPI_Comm_size, and 1 MPI_Finalize statement.							

Each line that adds to the conceptual complexity score performs some parallel-related call. It should be noted that this does not take into account variable declarations that are used for parallel work. For example, the MPI matrix multiply code uses A_{local} matrices. The declaration of these are not counted as parallel constructs. It is only when they are used in parallel calls that they are counted, such as when they are parameters.

Each of these construct are summed together to produce a parallel conceptual complexity score for each benchmark. The score illustrates the programmability of a parallel application. A lower score means a less complex parallel program, and better programmability. Usability is positively correlated with programmability, so a lower score translates into better usability. As an application increases in size and complexity, the parallel conceptual complexity of a program becomes increasingly important [92].

3.2.4 Code Development Time

Code development time measures how long it takes to develop each specific kernel. The code development time for an application is a critical measure of productivity. More programmability in a parallel model will lead to faster programming.

Unfortunately, specific time integrals were not measured in the process of this work. However, general remarks about development time are formulated for these kernels, and are found in the "Results and Analysis" section.

3.2.5 Other Metrics Not Implemented Here

The metrics described above document important characteristics about parallel conversion for codes. There are other metrics that are not implemented in this research, but could also serve as useful studies for languages in a holistic way.

These include studying a model's development resources in the form of documentation and development tools, studying its debugging, visualization, and analysis tools, its compiler support, and its acceptance by the general HPC user community. The scope of these is beyond this research paper, but they are still important measures of a parallel model's holistic evaluation. They are also large factors in a model's mainstream usage.

3.3 Coding Style and Expressivity Standards

To help maintain consistency between kernel implementations and different models, a series a of expressivity standards will be enforced on the code. A coding standard makes the implementation of a code follow a similar style, and is used to facilitate easier analysis and metric scoring. The standards adhered to for development are:

1. No more than one variable declaration per line.
2. No more than one statement per line.
3. Brackets, **do**, and **end** keywords (used in Fortress) each reside in their own line.
4. Blank lines, comments, and non-required tabs and spaces will not add to LoC, NoC, or CpL.
5. "Built ins," or functions meant to make programming easier to implement will be used if such expressiveness exists, such as **foreach** calls or or collective communication calls.
6. Programs will be written in as straightforward a way as possible. When possible, the easiest form of parallelization will be used. This "quick and dirty way" will be noted if it is revised at all, and both implementations will be noted. (It should be noted that this is only done to Titanium programs.)
7. The code will be structured in a "pretty-print" style, with generally accepted indentions and spacing.

These standards are being implemented for a three-fold reason.

- They maintain a uniform approach to code development across three different programming and semantical models.
- These standards emulate the "quick and dirty way" of programming, forsaking tedious optimization to see if enhanced productivity results.

- The standards are being implemented in the hope of ensuring a fair treatment of the benchmarks, that they are all represented at an equal level of programming sophistication.

3.4 Kernel Specifications

The computational kernels selected for implementation are a matrix multiply and a matrix transformation. Each of these applications illustrate an important aspect of scientific computing, computation and communication. They are naively solved through simple parallel means to illustrate the different communication paradigms between MPI, Titanium, and Fortress.

3.4.1 Matrix Multiply

Matrix multiplication is one of the most fundamental operations in linear algebra, and one of the most fundamental problems in scientific computing [93]. It serves as the main building block for many different algorithms. As HPC deals with scientific calculations, matrix multiply is widely used in parallel algorithms.

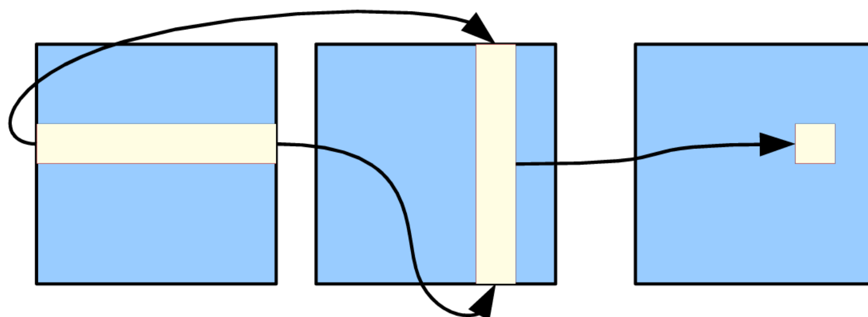
A matrix multiply takes two matrices, A and B, where A has an equivalent number of columns to B's number of rows. If A is an $m * n$ matrix, and B a $n * p$ matrix, the resulting matrix will have dimensions $m * p$. If $A * B = C$, then each point in C, denoted as C_{ij}, may be represented as

$$C_{i,j} = \sum_{k=1}^{n} = A_{i,k} * B_{k,j} = A_{i,1} * B_{1,j} + A_{i,2} * B_{2,j} + ... + A_{i,n} * B_{n,j}$$

By using a set of three **for** loops to iterate through the matrices, a naive implementation yields a time complexity of O(n^3). There are many other matrix multiplication implementations besides a straightforward solution [94, 95, 96, 14], including Strassen's algorithm [97] and Winograd's variation [98, 93], which both use clever techniques to reduce time complexity. However, this paper implements the easy-to-understand method described above.

For performing the multiplication in parallel, each point C_{ij} may be computed independently by a different processor. Therefore, each process may work on computing a subset of C_{ij} points.

Figure 3: Matrix Multiplication : A * B = C



If **for** loops are parallelizeable, then each point C_{ij} may be trivially partitioned to processes. If not, the simplest way to distribute work is by a row or column-deconstruction algorithm, which divides up work evenly between processes. Using row-deconstruction, the matrix (for example A) is broken up into chunks defined by an interval, and each process performs work upon a unique chunk (called an Alocal matrix). Each A_{local} would be of size $interval * number of columns$. Each process is then responsible for computing a C_{local} matrix, from matrix multiplication of $A_{local} * B$.

To solve a chunk of matrix A, a process needs only to have access to that particular submatrix (A_{local} matrix), and all of matrix B. For performing matrix multiplication using $A_{local} * B$ matrices, the matrix must distribute a unique chunk to each process' A_{local} matrix, and distribute all of B to each process. Once this has happened, local computation may proceed in parallel without intraprocess communication. As local computation completes, each chunk's result matrix may be gathered together into a result matrix C. Thus, this kernel may utilize bulk communication at the beginning and end of computation, with parallel tasks running independently in between.

For computing the matrix multiply kernel, a number of simplifications were made for the purpose of more easily discernible code, and to simplify overhead. Square matrices are used, and the number of processes run must be a multiple of 2, to ensure that rows are distributed correctly. The specific steps taken in each implementation are given below.

MPI Implementation This implementation of a matrix multiply utilizes three collective MPI calls- **MPI_Broadcast**, **MPI_Scatter** and **MPI_Gather**. The basic steps in the MPI program are:

1. The root process allocates space for matrices *A* and *C*.

2. All processes allocate space for matrices A_{local}, C_{local}, and *B*.

3. The root process populates matrices *A* and *B* with values.

4. Matrix *B* is broadcast from the root process to all processes.

5. Matrix *A* is scattered from the root process among all processes. Each copy of A_{local} receives part of *A* from the root.

6. The A_{local} matrix is multiplied with matrix *B* in each process, resulting in a C_{local} matrix for each process.

7. A collective gather call takes each C_{local} and gathers it into matrix *C*, residing in the root process.

Titanium Implementation Like MPI, Titanium allows use of copy operations to distribute data structures among processes. However, using a global address space (GAS), we can simply view the matrices *A*, *B*, and *C* as global. (They are implicitly global if not specified as local.)

There are two way this problem may then be solved- through GAS-only programming, and through data copying between processes. With the "naive" GAS-only model, each process instantiates a pointer for matrices *A*, *B*, and *C* and then points them to the global matrices *A*, *B*, and *C*. A process only solves for part of *C* by solving a certain number of rows from matrix *A*.

Our basic outline of steps in this code is:

1. All processes create pointers *A*, *B*, and *C*. (At the moment these are all null pointers.)

2. The root process creates global 2d arrays for matrix *A*, *B*, and *C* and populates them.

3. A broadcast call makes the root process's *B* pointer pointed to by all processes' *B* variable. This is repeated for matrices *A* and *C*.

4. Each process performs matrix multiplication on a subset of rows of *A*, writing values directly into *C*.

This is a naive approach to code since it uses a global address space only. On shared memory, this application should perform well. However, for distributed memory, Titanium documentation gives methods for copying a global array to local demesnes. A real implementation meant for distributed memory would certainly do this. Because of this, the code will be modified to **copy** array B to all processes, and to copy parts of A into local matrices A_{local} for each process. Local computation will then proceed independently and sans interprocess communication, much like the MPI code. The local results will be stored in C_{local} matrices, which are gathered together at the end of computation to produce a final matrix C.

Implementing this requires the creation of additional matrices, A_{local} and C_{local}. A_{local} is for copied parts of *A* C_{local} matrices are copied to the root process at the completion of each process's local computation. Thus, our basic steps are:

1. All processes create variables *A*, *B*, *C*, A_{local}, and B_{local}.

2. The root process creates 2d arrays for matrix *A*, *B*, and *C*. It then populates matrices *A* and *B* with values.

3. A broadcast call makes the root process's *B* variable pointed to by all processes. This is done for matrices *A*, *B*, and *C*. This is a necessary step to implement the array copy, otherwise processes with null pointers will point to uninitialized arrays.

4. A portion of matrix *A* is copied to the A_{local} matrix in each process's local memory.

5. All of matrix *B* is copied to each process's local memory.

6. Each process performs matrix multiplication of $A_{local} * B$, the results are stored in the local matrices C_{local}.

7. C_{local}s perform remote writes, copying the local matrices on each process to the correct part of *C* in the root process.

Fortress Implementation The Fortress code for matrix multiply is implicitly parallel. Since **for** loops are by default parallel, each for loop is evaluated in parallel by implicitly created threads of execution, based upon compiler discretion. Therefore, the Fortress code simply contains a triplet of loops, one to step through the rows of *A*, one to step through the columns of *B*, and one to step through the individual points being added to *C*. To prevent concurrent writing to a point C_{ij} in the matrix, the built-in **atomic...do** statement is used. One could presumably do an optimization using data *distributions*. These were not supported at the time of this research, so this is not an option. So the triplet of **for** loops was used.

The Fortress steps are simply.

1. Allocate matrices *A*, *B*, and *C*.

2. Use implicitly parallel **for** loops to step through the matrix and compute each point, storing the results in matrix *C*.

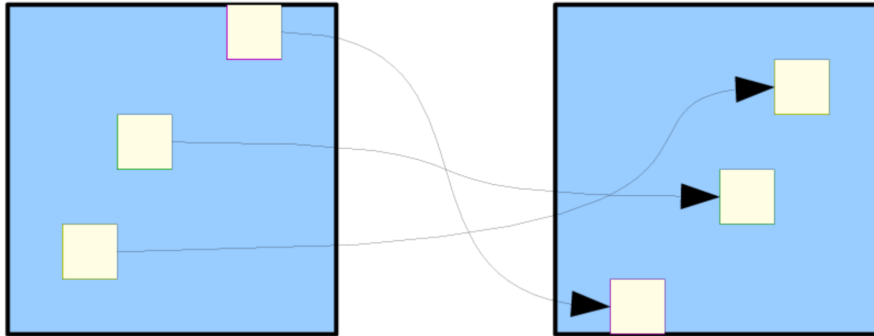
3.4.2 Matrix Transform

Like matrix multiplication, the matrix transform code illustrates an important concept in HPC. The matrix multiply code can be heavily computation intensive, with the MPI and refined Titanium code both communicating only at the beginning and end of the program. In contrast, this transform code places a heavy emphasis interprocess communication. This code is written to emulate a communication based application, that must communicate at every step to achieve computation.

The matrix used for the computation, A, is square like in the multiply kernel, with size *N**N*. To transform an individual point $A_{i,j}$, the value can exchanged be switched with the value $A_{N-1-i,N-1-j}$ (assuming indexing is from 0).

The emphasis on this matrix transformation kernel is in constantly sending back-and-forth data between processors. This is a kernel that relies entirely on communication. For Titanium, the processes may directly view the matrix as global and get from/put to it. For MPI, the matrix will be divided up into row-spanning chunks among processes and distributed as submatrices. Points

Figure 4: Matrix Transform



in one chunk will be exchanged with points in another chunk. Because of the way the matrix is partitioned, this communication will require back-and-forth communication. This communication will be pairwise in a "ping-pong" pattern, as each pair of processors exchange matrix values back and forth among their local matrices. When all matrix values have finished transposing among submatrices, the submatrices will be gathered together back on the root process.

For simplicity and to ease bounds checking, the matrix is an NxN square. The number of processors must be a power of 2, to allow for correct computation.

MPI Implementation For MPI, the matrix is scattered in intervals of rows among processes, similar to the matrix multiplication. The A_{local} matrices are comprised of a set of rows from A in a logical ordering; process with rank 0 gets the first chunk of rows, process 1 the second, etc. To transform an individual point $A_{i,j}$, it must be switched with point $A_{N-1-i,N-1-j}$. With the scatter partition breaking up rows, every call will require retrieving a remote value, and sending a remote value back to another process. The communication itself will form a kind of ping-pong, back-and-forth pattern between processes. Process 0 will communicate with process N-1-0, process 1 will communicate with N-1-1, and so on.

Once all values have been transformed, the matrix will be gathered back to matrix A in the root process. Thus, our basic steps are:

1. All processes allocate space for A_{local} submatrices.

2. The root process allocates and populates A locally.

3. A is scattered among processes, giving each process a filled A_{local}.

4. Processes exchange data back and forth to transform the matrix.

5. The transformed submatrices are gathered back to the root process.

There is a different way to implement this algorithm that must be mentioned. A smart scattering of the matrix would send row i and row N-1-i to the same process. Value A_{ij} could then be switched with value $A_{N-1-i,N-1-j}$ without interprocess communication. The communication savings would be substantial, and the transform would perform much faster.

However, this way of implementing is purposely avoided. This is because we wish to illustrate productivity and performance where constant communication is forced, as stated above. The point of this kernel is in requiring communication at every step. Therefore, while this simple transform may be more efficiently implemented, we are requiring an algorithm that does back-and-forth communication.

Titanium Implementation The Titanium implementation of this can be done by having one global array. Each process may transform a certain set of rows with another set of rows by transparently accessing and exchanging data in the global array. For this, we are allowing a processor to perform point switches both ways between a point A_{ij} and $A_{N-1-i,N-1-j}$. That is, a process may transform all values in row i, and in row N-1-i. Communication will still happen by remote reads and puts to the global matrix. Thus, our basic steps are:

1. The root process allocates A as a global array and populates it.

2. All processes may directly do transform operations on A on their specified rows.

Like the MPI code, this transform could be more "smartly" done with copying. Chunks of the matrix could be copied to each processor, and transformed in the same way as the MPI code. However, this does not force communication at every step, and so is avoided.

Fortress Implementation Like matrix multiply, this Fortress kernel makes use of implicitly parallel **for** loops to elicit parallelism. By only looping over upper half of the matrix, the values in the top may be exchanged with values in the bottom. Therefore, using two implicitly parallel loops, the top half or rows, and all columns may be traversed. Each point in this upper half will be exchanged with a value in the lower half, in an **atomic do** statement to ensure concurrency. Since Fortress's loops are implicitly parallel, and may execute out of order, the distribution of tasks to processes is left up to the compiler.

Our basic steps here are:

1. Allocate and populate matrix A.
2. Using two loops, traverse the top half of the matrix, exchanging values with the bottom half of the matrix.

3.5 Closing Methodology Notes

In the following section we present the results of implementing and benchmarking the kernels. Additionally, we will holistically analyze the results of code development and compare all the results in an evaluative sense, determining which language gives the most performance, and which language is the most programmable.

4 Results and Analysis

This chapter presents results and analysis of performance and programmability benchmarking. Both MPI and Titanium were extensively tested on a test cluster at the University of Arkansas, Fayetteville. At Fortress's current state of language development and implementation, complete performance testing was not possible. Some basic runtimes were done; and they ruled out the language from any real evaluation. Even though Fortress was not seriously tested, the computational kernels were written, and a limited set of programmability benchmarks were applied to it. MPI and Titanium were evaluated both in performance testing, and in terms of programmability. This allows for a complete holistic characterization for these languages. Testing to the limits of memory was not done though, the largest matrix sizes that were used were 2048x2048. This chapter presents the results of both the benchmarks and the holistic characterizations.

For the multiply code, there are different codes compared. Since Titanium code has been written in both "naive" and "real" implementations, Titanium code is distinguished as either Ti-Naive, or Ti-Real. The MPI code is simply referred to as MPI.

4.1 Hardware Setup and Testing Setup

All runs were performed on "tusk.uark.edu." This is a heterogeneous 16-core cluster consisting of two 8-core nodes, called "Tusk" and "Tusk-Sun." This allows shared memory jobs of up to eight cores on either node, or up to sixteen cores using both nodes for distributed memory jobs. Tusk is an 8-core system consisting of two Intel Xeon quad core L5320 1.86 GHz CPUs. Each processor core has 1MB of cache. Tusk-Sun consists of four dual-core AMD Opteron 8218 Processors. Each CPU is rated at 2.6 GHz, and the system has 1 MB of cache per processor. Also, the cluster is connected with a single 8-port Gigabit Ethernet switch. Each node runs Ubuntu Linux Server edition 7.10. MPICH version 1.1.2 is installed. For some programs the Moab batch scheduler was used to execute jobs.

Matrix Setup For each benchmark, several sizes of square matrices were used. The matrices had sides of length equal to 128, 256, 512, 1024, or 2048. This was hard-coded into the syntax to make it easier to perform productivity analysis. Executables were generated for each needed matrix size. Most shared memory runs were done on Tusk. When doing distributed runs, the cores were divided evenly between Tusk and Tusk-Sun, in a 1 to 1, 2 to 2, 4 to 4, or 8 to 8 ratio.

The matrices used "double" numbers, each of which take up 8 bytes of memory. The size of memory of a matrix could then be represented as $length * width * 8$ bytes. Matrices of size 128 and 256 were small enough to reside in cache.

Tusk and Tusk-Sun make up a heterogeneous system. They have different CPUs with different cores, cache, and clockspeeds. This is valuable for testing real-life scaling as the different systems perform computation at different speeds. Running a distributed memory job using Tusk and Tusk-Sun means that some part of computation will finish at a faster rate than the other, and be required to wait for the slowest denominator to catch up to barriers or communication. To evaluate the difference between Tusk and Tusk-Sun running shared memory jobs, a set of runs at matrix sizes equal to 256 and 1024 were done on Tusk-Sun. For distinguishing when code is run on Tusk and Tusk-Sun, the prefix T- or TS- is used on code. A matrix of size equal to 256 could fit totally in cache on both systems.

Titanium Compilation Settings Titanium code invoked the Titanium compiler by using the tcbuild command. The Titanium compiler was version 3.202 and was built with GCC 4.1.3. For shared memory, code was compiled with the **backend=smp** flag, which compiles specifically for shared memory. For distributed memory, code was compiled using the **backend=mpi-cluster-smp** flag, which uses MPI for intranode communication. The GASNet message layer [71] is not currently available on Tusk. It should be noted that the tcbuild command was not invoked with either **–optimize**, nor **–lqi** as options, as the Titanium compiler build was not done with local qualification inference enabling. Code was run either using PBS scripts, or from command line using an **mpirun.mpich** command. Performance was identical using either method to execute the

application.

MPI Compilation Settings MPICH version 1.1.2 was used. MPI files were compiled with **mpicc**, and run using the **mpirun.mpich** command, or by PBS script. PBS scripts were used for most MPI distributed memory jobs.

Fortress The Fortress interpreter was called directly by use of the *fortress* command. This parses and evaluates the program, albeit slowly compared to Titanium and MPI's execution. At the time of this research, the limitations imposed by this made performing scalability and performance testing impossible. Also, it was impossible to test process parallelism as the ability to manually control processes was not then implemented in the language core. To that end, the Fortress code is still presented in the index. A limited productivity characterization and usability score for Fortress are calculated, but with Fortress's rapid rate of change, these characterizations may not be valid for long.

4.2 Benchmark Results and Analysis

In this section, we present all code benchmark results and analyze the performance of each. Both shared and distributed memory results are presented for 2, 4, and 8 cores. Additionally, benchmarks with 16 cores from distributed memory testing are presented. Shared memory runs were done on the node Tusk, except for runs distinguished in Section 4.2.3 and 4.2.6 as being done on the node Tusk-Sun.

4.2.1 Shared Memory Multiply Results

Two Titanium programs were written to do matrix multiplication, a "naive" and a "real" application. These programs are referred to as Ti-Naive and Ti-Real. These programs are compared with the matrix multiply MPI code in the Figures 5 through 9.

As expected during shared memory benchmarks, "Ti-Real" and "Ti-Naive" have similar run-

time on Figures 5 though 9, showing nearly identical performance and scaling. In an shared memory environment, there is only one memory region, and all processes allocate out of this shared pool [67]. Thus, for all processes to get data from matrix A, and to put data in matrix C, only a simple load is required. On shared memory this is a local operation. Note that this is not the case for distributed memory. See the section 4.2.2. Thus the Ti-Naive code is not penalized for its "global" references to arrays , as they are translated into local references.

The Ti-Naive code is slightly faster than Ti-Real on all shared memory runs, as shown in Figures 5 through 9. The reason for this is that Ti-Real does copy operations between matrix A and A_{local}, B and (local copies of)B, and C and C_{local}. These copies, while migrating data that is local, still take a small amount of time. The separation between Ti-Real and Ti-Naive exhibits a consistent scale, and thus grows slightly as the matrix size increases from 128∗128 to 2048∗2048. For each copy operation, $N * N * 8$ bytes are copied between local memory addresses, where N=length of the matrix. Each process will do this twice in a Ti-Real application, once to copy data from A to A_{local} and once to write data from C_{local} to the root's C matrix. Additionally, an extra $N * N * 8$ bytes are copied in copying B to B from the root process to "local" B's for each process. The other computation/communication between Ti-Real and Ti-Naive takes the same amount of time.

Figure 5: SM Multiply of length 128x128

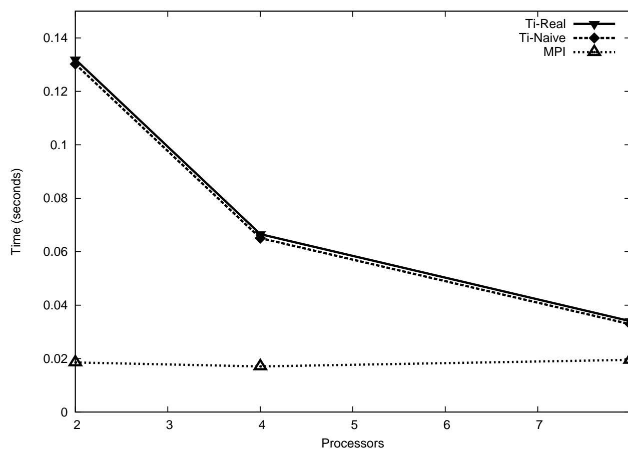
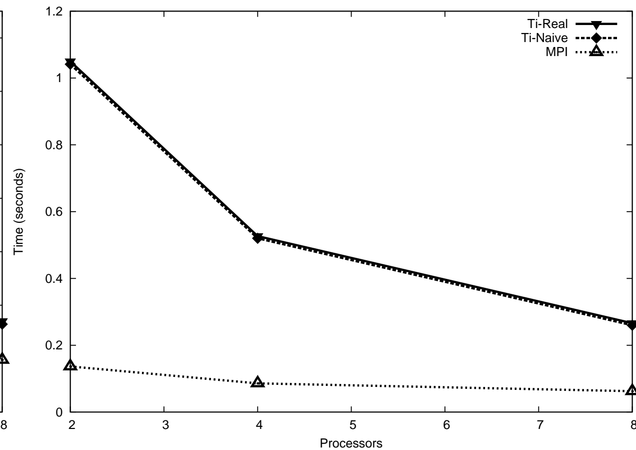


Figure 6: SM Multiply of length 256x256



Shared Memory 128x128 Figure 5 shows shows multiplication times of two $128 * 128$ matrices. The Ti-Real and Ti-Naive code exhibit extremely similar runtime and scaling, nearly 1.97 times as processors double. The MPI code does not showing any scaling at all. The matrix sizes are so small that the time spend allocating local matrices and scattering the matrix among them overrides any gains achieved by parallel computation.

Each Titanium code in Figure 5 runs more slowly than the MPI, approximately 6 times slower for 2 processors, and 1.5 times slower for 8 processors. Since the MPI code doesn't scale at all, the gap between execution time gets smaller as Titanium processors increase.

Shared Memory 256x256 Figure 6 shows the $256 * 256$ matrix multiplication. This matrix still fits in cache on Tusk. The runtimes are still extremely fast because of this. The MPI scalability is moderate as well, 1.59 times from 2 to 4 processes, and 1.37 times from 4 to 8 processes. The Titanium code again shows nearly 2x scalability, going from slightly over a second to almost .2 seconds from 2 to 8 cores. According to the O(n^3) complexity, this code should take 8 times as long to compute as the 128x128 matrix code. This is verified by these results. Despite Ti-Real and Ti-Naive code also showing almost exactly the same runtimes, they are still about 7 to 4 times as slow as the corresponding MPI code for a given amount of cores.

Figure 7: SM Multiply of length 515x512

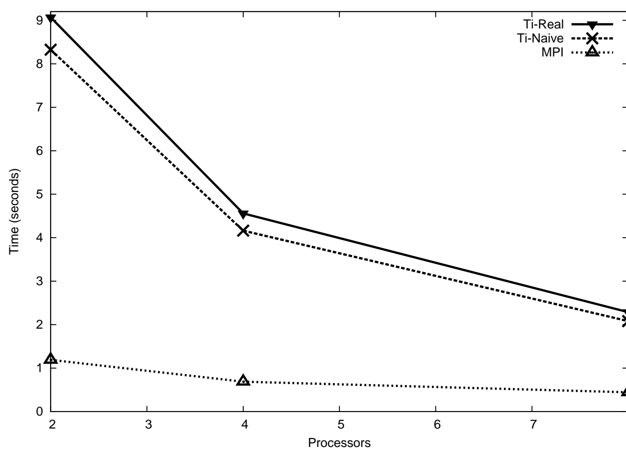
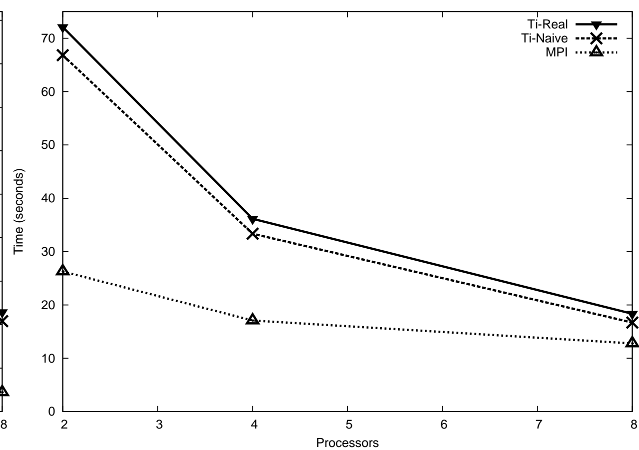


Figure 8: SM Multiply of length 1024x1024



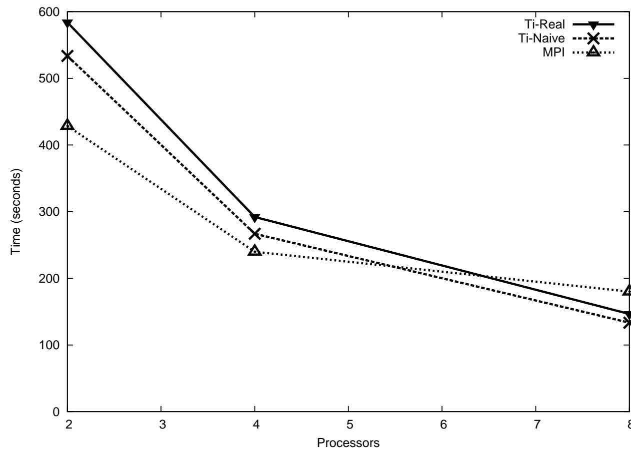
Shared Memory 512x512 Figure 7 shows results of matrix multiply in shared memory when the matrix is too big to fit in cache. MPI runtime on 2 processes is increased to over a second, and times for Titanium are over 8 seconds. This is the first size for shared memory that shows MPI scaling as processor size increases. Runtime for MPI code goes from almost 1.2 seconds to .7 to .4 seconds. This is a little over 1.7x scaling as processors double. The Titanium code still scales like expected, at nearly 1.99x scaling as processors double. According to the $O(n^3)$ complexity, this code should take 8 times as long to compute as the 256x256 matrix code. This is true for both Titanium and MPI, as they take about 8 times longs to compute. At 2 processes, the MPI code runs around 7 times as fast as the Titanium code, and around 5 times as fast for 8 processes. The better scalability of the Titanium codes closes the ratio a little bit here.

Figure 7 is the first graph to show the distinction between the Ti-Naive and Ti-Real runtime that is easily discernible. In the Ti-Real code, the cost of copying arrays that are larger than cache size hurts its performance, and gives a little bit of a gap between the Ti-Real and Ti-Naive code.

Shared Memory 1024x1024 Figure 8 exhibits a large jump in timing from Figure 6. The Titanium timings range from around 70 seconds for 2 processors, to just under 20 seconds for 8 processors. The MPI code likewise goes from 26 to 12 seconds runtime. Ti-Real and Ti-Naive experience nearly identical scaling of 1.98x processes double, and take nearly 8 times as long to compute as the 512x512 matrix, thus verifying the $0(n^3)$ complexity. The MPI code experience scaling of 1.53x from 2 to 4 cores, and 1.34x from 4 to 8 cores. The Titanium runs are between 2.7 times and 1.4 times as slow as the MPI code across the same number of cores.

Shared Memory 2048x2048 Figure 9 shows the longest runtimes for shared memory matrix multiplication. The Ti-Real code goes from a runtime of around 584 seconds to about 147 seconds, exhibiting 1.98x scaling as cores double. The Ti-Naive code goes from 534.5 to 133.4 seconds, and shows an even better 1.999x scaling as cores double. The MPI code goes from runtimes of 429 to 249 to 180 seconds as cores double, scalings of 1.79x and 1.33x. One very interesting point here is that Titanium's code actually runs faster than MPI's code for 8 cores. With 2 cores, the

Figure 9: SM Multiply of length 2048x2048



Titanium code runs approximately 1.25x-1.36 times as slow as the MPI code, or between 100 and 150 seconds slower. At 4 cores the code is between 55 and 30 seconds slower. At 8 cores the Titanium code is between 40 and 50 seconds faster, or 1.23 times and 1.35 times as fast. This is the only time that this happens in all shared memory multiply runs. As Titanium's scaling is still almost 2x as processors increase, it is able to overtake the MPI code in runtime. The Titanium code also continues to show consistent verification of the $O(n^3)$ complexity, as the code takes approximately 8 times as long to compute as the 1024x1024 code.

One note will be made here about the difference between Ti-Real and Ti-Naive scaling. With 8 cores doing this problem, the Ti-Naive code runs 1.09 times faster than the Ti code.

Shared Memory Multiply Notes For all shared memory runs, Titanium showed excellent scalability, around 1.98x to 1.99x for all runs. The Ti-Naive code showed scalability near 1.999x as cores doubled, and thus was slightly faster at computation as problem size increased. With 8 cores doing 2048x2048 matrices, the Ti-Naive code took about 533 seconds and the Ti-Real code about 583 seconds. The average speedup of Ti-Naive code over Ti-Real code was 1.09x- this speedup is consistent over different problem sizes and cores. It is interesting note that, for shared memory jobs, the more naive GAS-only approach is the programming style that generates faster computation, even if only slightly by a factor of 1.09x.

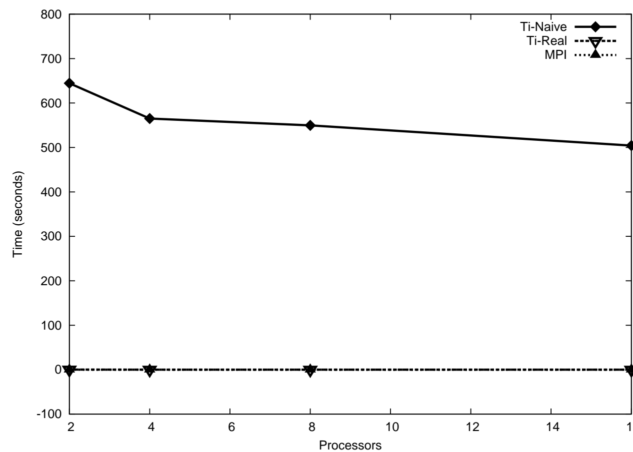
When the matrix sizes were above 256x256, the MPI code exhibited scaling from around 1.7x to 1.3x. The fact that MPI could not scale as efficiently as Titanium meant that when the problem size became big enough, and there were enough processes, that the Titanium code would be faster that the MPI code. For our runs, this only happened using 8 cores with matrices sized 2048x2048. It is hypothesized that Titanium will continue to show similar scaling with larger matrix sizes and more processes, as long as memory access is uniform.

4.2.2 Distributed Memory Multiply Results

The matrix multiply kernels were made to be run on distributed memory as well as shared memory. The runtime graphs and analysis are presented in this section. The first analysis deals with the Ti-Naive code on distributed memory. The code exhibited such bad performance it was decided to discontinue running it for shared memory, as runtime for larger problem sizes would quickly cause runtime to get out of hand- taking hours, days, weeks, and even months to perform a single matrix multiply.

This was the only set of distributed runs that the Ti-Naive code. Although Ti-Naive and Ti-Real (called Titanium (Tusk) in figures) have nearly identical performance on shared memory, using a global array is shown to be very costly at times.

Figure 10: DM Multiply of length 128x128 w Naive Titanium Code



DM 128x128 using Ti-Naive We will discuss this set of runs first, as this is the only time that Ti-Naive code was tested on distributed memory and message passing. The Ti-Naive code performs very well and exhibits almost 2x scalability in shared memory. However, on distributed memory its use of GAS programming cripples solution's computation performance.

The Ti-Naive code exhibits runtimes from almost 650 seconds for 2 processors to just over 500 seconds with 8 processors, and exhibits no better scalability than 1.14x as cores double (this from 2 to 4 cores). The Ti-Real and Ti-MPI code, discussed in the next paragraph section, both complete computation in under .3 seconds for the same problem.

Although no extensive analysis of the remote calls made is done for all matrix sizes, we will note on it here, due to the extraordinarily bad performance. A $128*128$ matrix has 16,384 elements in it. For solving a point in the result matrix, C_{ij}, a process through a row of matrix A and a column of matrix B, adding the result to C_{ij}. This requires N accesses to get values from the A row, N accesses to get values from the B column, and an additional $2N$ accesses to C_{ij}, one to get the current value, and then one to put the modified value back into it. Therefore, for multiplying 2 $N * N$ matrices, Ti-Naive code requires $N * N * N * N * 2N$, or $2N^4$ accesses. For our $128 * 128$ matrix running on tusk, there are 536,870,912 global calls. For the region that the array resides on, this global call does not take as long as a truly memory-remote call, but there is still overhead associated with doing the global call at all. This is clearly unacceptable for parallel computation.

DM 128x128 The distributed memory run times of Ti-Real and MPI are presented in Figure 11. The matrices size here is small enough to reside entirely in cache, and runtimes are extremely fast. The MPI code exhibits no scalability at all between processes. The overhead associated with copying the data overwrites any performance gains due to being performed in parallel. The Titanium code shows some scaling from 2-8 cores, although at 16 cores it faces into the same copying overhead that MPI does, and its scalability suffers.

DM 256x256 Figure 12 shows the results of matrix multiplication with sides of length 256. At this size, the Titanium code shows scaling above 1.9x from 2 to 4, and 4 to 8 cores. From 8 to

Figure 11: DM Multiply of length 128x128

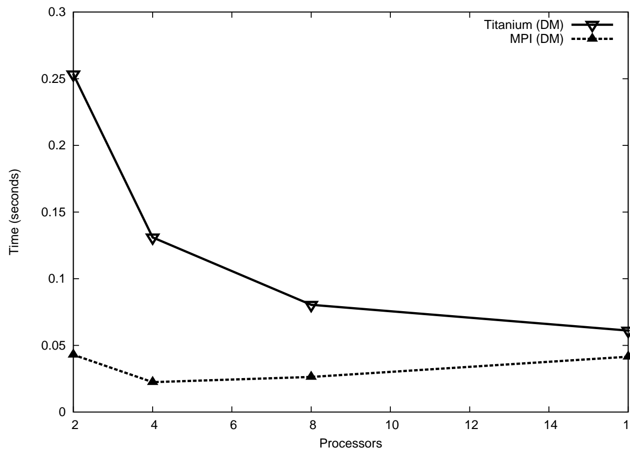
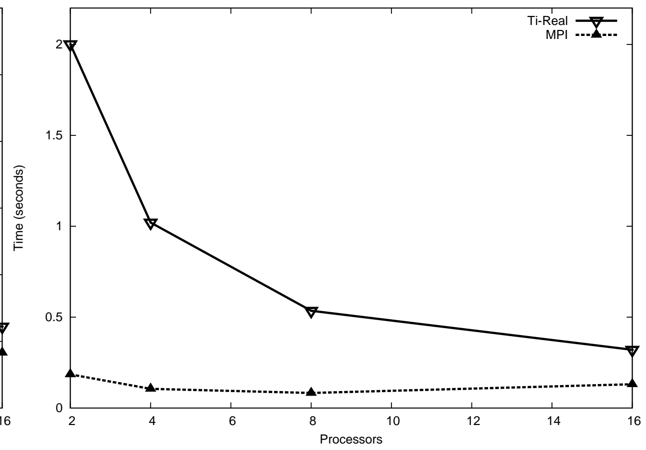


Figure 12: DM Multiply of length 256x256



16 processes, its scaling goes down to 1.66x. The MPI code runs scales 1.75x from 2 to 4 cores, and doesn't exhibit scaling after this as cores increase. Running at 2 cores, the Ti-Real code is 10.8 times slower than the MPI code. Since the MPI code does not scale while the Titanium code does, the gap between the Titanium and MPI goes from 1.81 seconds to .1 seconds, as on 16x cores the Titanium code is 2.45 times slower than the MPI code. The Titanium code here takes approximately 8 times as long to compute as the code for the 128x128 matrix, although the MPI code only takes approximately 4 times as long.

Figure 13: DM Multiply of length 512x512

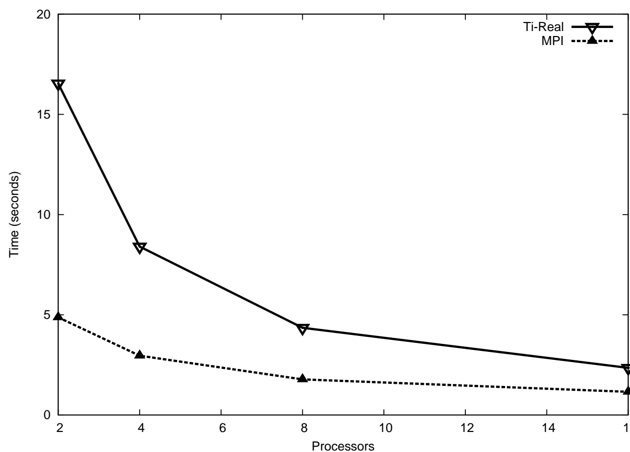
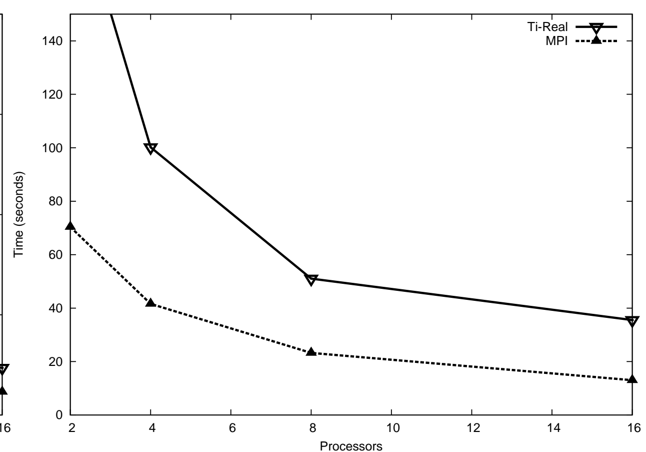


Figure 14: DM Multiply of length 1024x1024



DM 512x512 Figure 13 presents runtimes for $512x512$ matrices. The Titanium code exhibits scalability above 1.8x as cores double, with its highest being 1.96x from 2 to 4 cores. The MPI code exhibited scaling close to 1.6x as cores increased. At 2 cores, the Titanium code was 3.4 times as slow as the MPI code, this changed to 2 times slower at 16 cores, thanks to Titanium's better scalability. The Titanium code is approximately takes 8 times as long to compute as the 256x256 code, while the MPI code takes about 20 times as long.

DM 1024x1024 The largest distributed memory runs were done at matrix size $1024 * 1024$. The runtime graph is presented in Figure 14. Like all other distributed memory runs, the Titanium code is overall still slower than the MPI code. It still exhibits good scalability though, at a rate of almost 2x per core doubling, except when going from 8 to 16 cores where it only scales by a factor of 1.4x. The MPI code displays a speedup of between 1.7x and 1.8x as cores double. The MPI code is 2.86 times faster than the MPI code at 2 processes, and 2.72 times faster than MPI with 16 cores. The Titanium code takes around 8.75 times as long to compute as the 512x512 code. The MPI code takes around 14 times as long.

DM Multiply Notes Although Titanium showed consistently better scalability than MPI on distributed memory, but was still consistently slower overall than MPI. The scaling of Titanium in distributed memory was less for some larger process sizes, notably the $1024 * 1024$ matrix where scaling is only 1.4x when going from 8 to 16 cores. The respective MPI scaling at that size matrix from 8 to 16 cores is almost 1.8x.

4.2.3 Tusk/Tusk-Sun Multiply Results

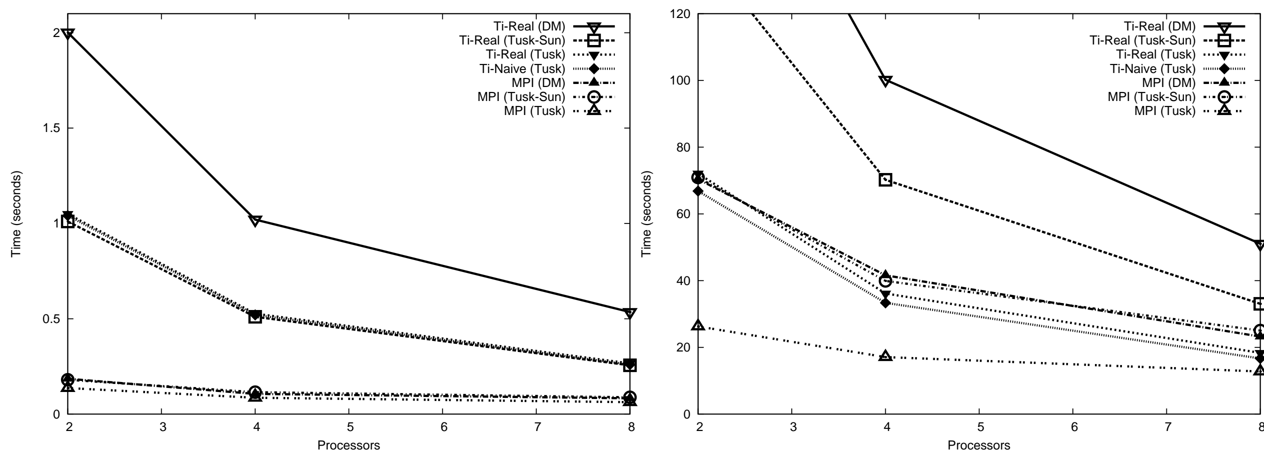
This section compares performance between Tusk and Tusk-Sun doing the multiply code. It is meant to illustrate how the kernels perform on different systems. All runs that were used for the shared memory datasets were based off of runs on Tusk. Distributed runs used half of their processes from Tusk, and half from Tusk-Sun. A subset of the matrices were run on Tusk-Sun as well, and they are presented here. The kernels were run with matrix sizes $256*256$ and $1024*1024$.

The $256 * 256$ sized matrices fit in cache on both systems.

Much previously analyzed data comes from the shared memory and DM runs for corresponding matrix size. These runtimes are presented and analyzed in detail in Sections 4.2.1 and . For clarification, runs on Tusk will be referred to as Ti-Real-TU, Ti-Naive-T, or MPI-TU. Distributed memory results are referred to as Ti-Real-DM or MPI-DM for the Multiply Code.

Figure 15: Tusk/Tusk-Sun Comparison Runs - Multiply with length 256

Figure 16: Tusk/Tusk-Sun Comparison Runs - Multiply Matrix 1024



Tusk/Tusk-Sun Multiply 256x256 Figure 15 presents a number of runtime plots. The only plots previously not analyzed are the ones for plotting TiReal and MPI on Tusk-Sun. These results are referred to as TS-Ti-Real and TS-MPI.

The MPI code across these three runs all are closely clustered together. The Tusk code is slightly faster than the DM- or TS- runs. There is a good reason for this. Since the multiply code is rather computationally intensive (as opposed to communicationally intensive like the transform code), processor computation is the limiting factor in the code. The multiply code distributes data at the beginning of computation and each process independently solves its part of the result. The big limiting factor then is processor speed.

At the end of computation a **gather** call takes all the answer matrices and compiles them to a final result matrix. Completion is not allowed until all processes finish computing their local

answers and copy data to the result matrix. Tusk-Sun computes data more slowly than Tusk as seen by the difference in results between TS-MPI and T-MPI. When doing the DM-MPI job, the local matrices solved by Tusk finish faster than the matrices finished by Tusk-Sun. Since the **gather** call must block for all processes to call before continuing execution, Tusk-Sun limits performance of DM-MPI and DM-Titanium to at least as slow as that of TS-MPI and TS-Ti-Real, respectively.

With small matrices, the difference in TS-MPI and DM-MPI is negligible. When MPI-DM communicates, it must send data over the network. When TS-MPI does so, it goes over the system bus. The data communication of the matrices is small, and so there is not a large difference in times between "local" data transfer that TS-MPI does, and the network communication of MPI-DM. Recall that with small matrices, the ability to scale is limited as well by communication overhead on shared memory and DM runs. This applies to Tusk-Sun runs as well. Since MPI-Tusk communicates over its own local bus, and has a faster computation ability, it has faster runtime.

The Titanium shared memory codes are all almost identical in runtimes. The Ti-Real-T, Ti-Naive, and Ti-Real-TU code all have negligible runtime differences between them. They exhibit similar scalability as the cores are doubled, and all three are notably faster than the Ti-Real-DM code.

Tusk/Tusk-Sun Multiply 1024x1024 Figure 27 displays results for the larger matrix sized tested on Tusk-Sun- $1024 * 1024$. Like the matrix size=256 results, the MPI-TU code runs faster than the TS-MPI and MPI-DM code. The TS-MPI and MPI-DM, which are limited by computation speed, still close together in runtimes and show similar scaling. The MPI-DM code is slightly slower than the TS-MPI code, as its communication is over a network, and not between local memory, as TS-MPI.

The TS-Ti-Real code is surprisingly slower than the Ti-Real-TU code, and very close to the Ti-Real-DM code.

There is a visible difference in runtimes between the Ti-Real-DM and TS-Ti-Real code. Both codes are limited by the computation speed of Tusk-Sun. The TS-Ti-Real code runs entirely on

Tusk-Sun and only uses its processors. The Ti-Real-DM code runs half on Tusk and half on Tusk-Sun, and uses a **barrier** to wait for all computation to finish. Since the Tusk-Sun computation happens slower than the Tusk computation, it must wait at that long to complete. The overhead of doing global calls as opposed to local copies is now seen, as the -DM code is between 20 and 40 seconds slower than the corresponding TS- code. The Ti-Real-TU code is faster than both of them, as it has faster local computation, and does its communication through local copies.

4.2.4 Shared Memory Transform Results

The matrix transform operation was done with two programs- a ping-pong communicating MPI program and a Titanium program using a global array. Both programs tested heavy communication between different processes and process memory.

The results for the shared memory transform code is below. The Titanium code relies on a global data structure that each process accesses and modifies. On the shared memory, these data accesses are memory loads and stores- extremely fast. Therefore the Titanium code is much, much faster than the corresponding MPI code for all matrix sizes and number of processes. It is difficult to see the runtimes and scaling of the Titanium code in these graphs. The exact timing measurements may be referenced in the Appendix to see validation that the Titanium code does scale.

Shared Memory Transform 128x128 Figure 17 shows the MPI and Titanium runtimes for doing the transform. The MPI code is much slower than the Titanium code- at 2 cores the MPI code is over 1000 times slower, and is over 300 times slower at 8 cores. This is because the Titanium code ping pongs messages through abstract global references. These are translated to simple loads and stores for the Titanium code, giving it runtimes of around 0.0005 seconds. The Titanium code does not scale between processes at this time, as the overhead with distributing the intervals makes the parallel evaluation gain negligent.

The MPI code, however, does benefit from scaling, exhibiting about a 2x speedup from 2 to 4

Figure 17: SM Transform - 128x128 Matrix

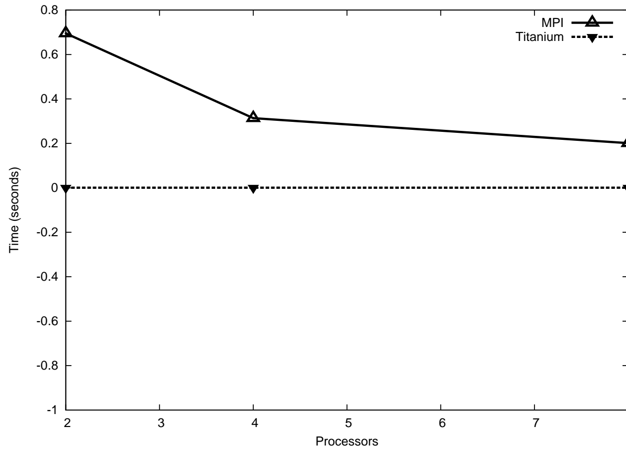
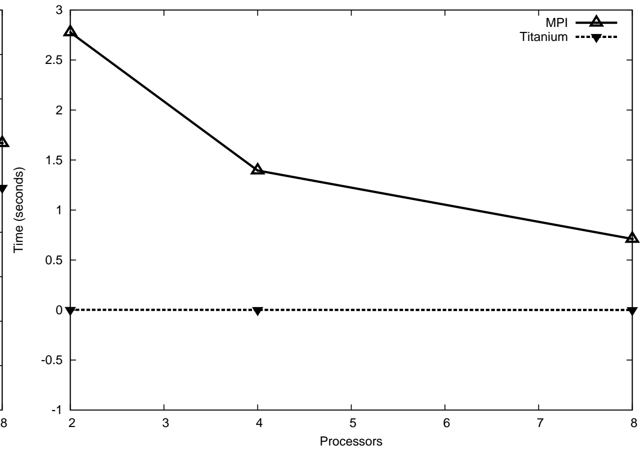


Figure 18: SM Transform - 256x256 Matrix



cores, and a 1.5x speedup from 4 to 8 cores. It should be noted however, that the overall runtime is so small, however, that these results may easily differ enough to skew the performance scaling.

Shared Memory Transform 256x256 The Titanium code is still extremely fast, between 1090 times faster for 2 and 4 cores, and 660 times faster for 8 cores. Although not discernible in the graph, the Titanium runtime does drop slightly, from .0026 to .0014 to .001 seconds, with speedups of 1.82 times and 1.31 times. The MPI code exhibits a 1.99 times and 1.96 times scaling. The Titanium code scales somewhat from 2 to 4 cores (a 1.82 times scaling), therefore Titanium code is still almost 1000 times as fast as the MPI code at 4 cores (specifically, .00140 seconds to 1.39565 seconds). The MPI code exhibits high scalability in this dataset, as doubling processors nearly halves the computation speed. The MPI code here takes 4 times as long to compute as the code for the 128x128 matrix, which is consistent with the $O(n^2)$ complexity of this problem.

Shared Memory Transform 512x512 Figure 19 shows the runtimes from transforming a $512 * 512$ matrix. The Titanium code is fast as expected, having 2x scalability, even with its limited runtime. The MPI code likewise exhibits scaling of near 2x and 1.62x for going from 2 to 4 to 8 processes. The MPI code takes 1200, 1000, and 1050 times as long as the Titanium program to complete. It also takes 4 times as long to compute as the 256x256 MPI code, which is consistent

Figure 19: SM Transform - 512x512 Matrix

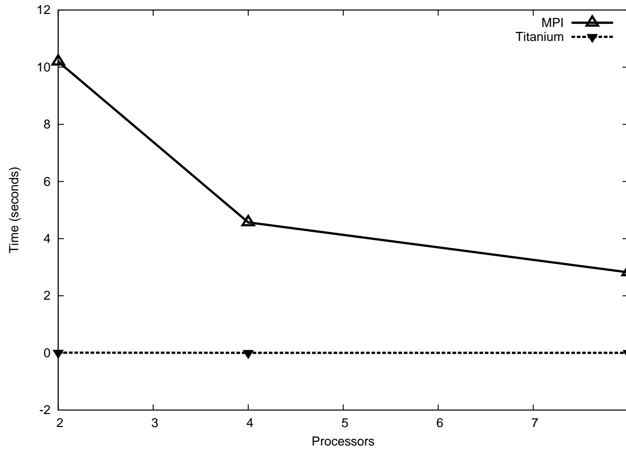
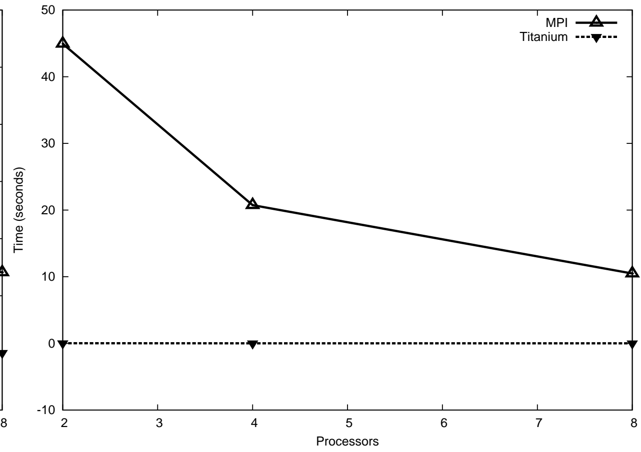


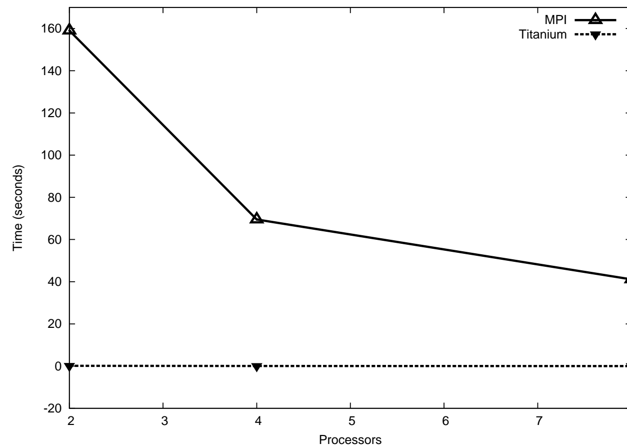
Figure 20: SM Transform - 1024x1024 Matrix



with the O(n^2) complexity.

Shared Memory Transform 1024x1024 Figure 20 shows the runtime for a $1024 * 1024$ matrix. The Titanium code is between 1400 times and 1200 times as fast as the MPI code, and exhibits 2x scaling as processes increase. The MPI code also exhibits very good scaling near 2x as processes are doubled. The scaling between 2 and 4 processes is actually 2.16x on MPI from 2 to 4 processes. The MPI code approximately takes 4 times as long to compute as the 256x256 MPI code, which is consistent with the O(n^2) complexity.

Figure 21: SM Transform - 2048x2048 Matrix



Shared Memory Transform 2048x2048 Figure 21 shows the largest matrix size transposed, $2048x2048$. Following all other matrix sizes, the Titanium code was extremely fast- between .13 and .03 seconds. The scaling was also almost perfectly 2x as processes increased.

The MPI code exhibited a 2.27x speedup from 2 to 4 cores. From 4 to cores, it had a 1.7x speedup. The MPI code was in general 1240 times to 1000 times times slower than the Titanium code. The MPI code also displayed consistent $O(n^2)$ complexity, taking approximately 4 times as long as the 1024x1024 code to compute.

Shared Memory Transform Notes The GAS ability of the Titanium code is fully exploited on shared memory, where the global references in this code are simply converted to local references. These references make the matrix transform code trivially fast, as data communication is optimized to be local. The MPI code cannot recognize this though. The standard is not designed to, and therefore all interprocess communication is explicit. The matrix must be decomposed between processes and explicitly transformed through back and forth communication, even though matrix copies reside in local memory reach of other. This is unfortunately a limitation of the MPI standard, and an advantage of the Titanium compiler.

The shared memory transform code demonstrates the ability of the Titanium compiler to optimize communication when jobs are run on shared memory. This makes this communication intensive benchmark run extremely fast on Titanium when global references are used. Unfortunately, remote calls take much longer in distributed memory.

4.2.5 Distributed Memory Transform Results

The Titanium and MPI code port seamlessly to distributed memory. The relative performances between the two kernels across matrix sizes and number of processes is now completely different. Whereas the Titanium compiler cleverly realized that the shared memory code could make global references local, distributed memory affords no such luxury. Global references to the global array are now truly global in Titanium. This therefore does a true distributed-communication test be-

tween Titanium and MPI. There are 5 graphs presented displayed the runtimes between the two kernels for the different matrix sizes, Figures 22 through 25.

Figure 22: DM Transform - 128x128 Matrix

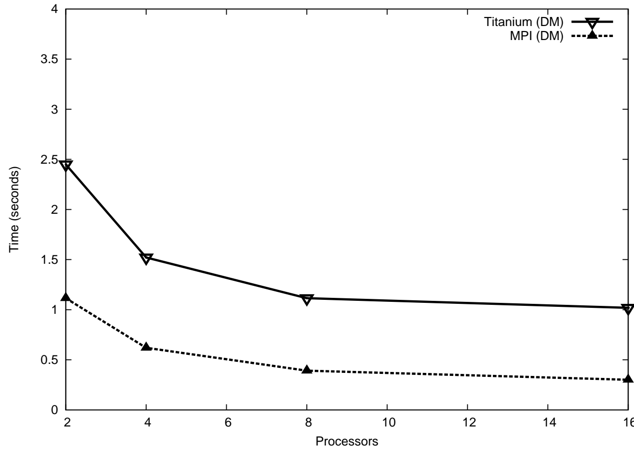
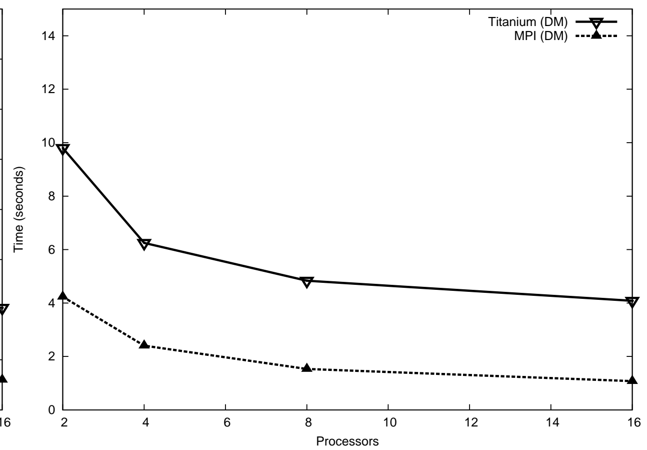


Figure 23: DM Transform - 256x256 Matrix



DM Transform - 128x128 Matrix The runtimes for a $128 * 128$ matrix are shown in Figure 22. The Titanium run codes are very different looking than they were on the equivalent shared memory job, even though the code is the same. The Titanium code is now approximately 2.2 times slower than the MPI code across cores, and exhibits decreasing scaling, 1.6x to 1.3x to 1.1x from 2 to 4, to 8, to 16 processes.

The MPI code exhibits moderately better scaling as processes double, 1.8x scaling from 2 to4, 1.6x from 4 to 8 and 1.3x from 8 to 16. This is because the MPI code partitions the array to pairwise communicating processes. The processes only communicate with each other to transpose their submatrices. The Titanium processes do accesses on the global array in the root process's memory region. There is contention for bus access here, and as processes increase, this affects scaling negatively.

DM Transform - 256x256 Matrix Figure 23 displays the runtimes for matrix transform of size $256x256$. Like the $128x128$ transform, the MPI code exhibits better runtime. It's scalability is also slightly better than Titanium as processes double, 1.75x to 1.56x, 1.57x to 1.3x, and 1.42 to 1.18x

Figure 24: DM Transform - 512x512 Matrix

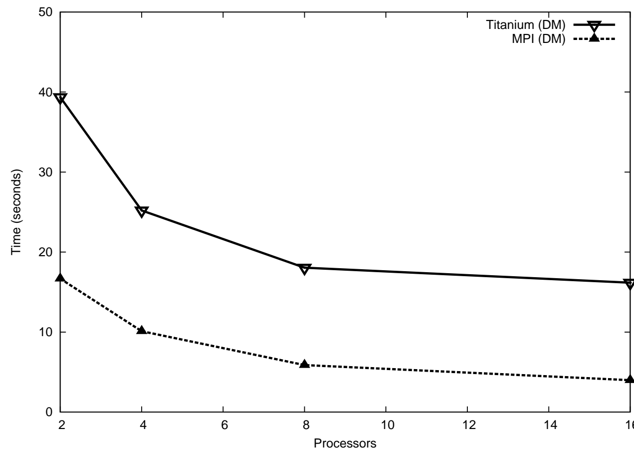
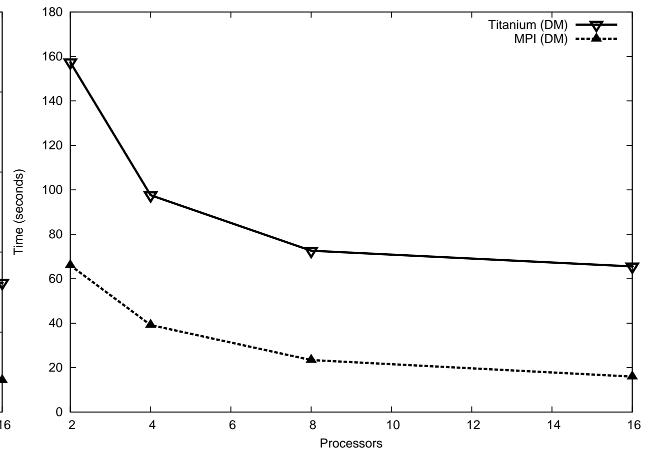


Figure 25: DM Transform - 1024x1024 Matrix

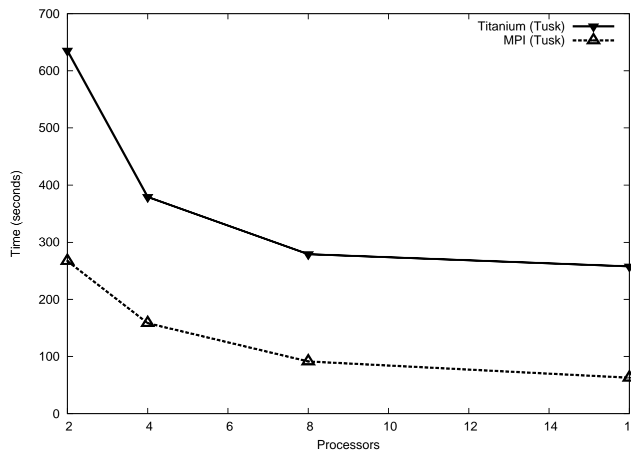


as processors go from 2 to 16. The MPI code is between 2.31 and 3.78 times as fast as the Titanium code, as more processors are added this increases slightly due to Titanium's lower scalability. Both codes justify the O(n^2) complexity of this problem by taking approximately 4 times as long to solve as they did for the 128x128 matrix.

DM Transform - 512x512 Matrix Figure 24 shows the transform timings for the $512 * 512$ matrix. The Titanium code shows scaling of 1.56x, 1.40 times, and 1.11x between core sets, and the MPI code scaling of 1.65x, 1.72x, and 1.47x. The Titanium is between 2.35 and 4.1 times as slow as the MPI code- this increases as cores go up. Both the MPI and Titanium code take approximately 4 times as long to compute as they did for the 256x256 matrix.

DM Transform 1024x1024 Matrix Figure 24 shows the runtimes for the the $1024*1024$ matrix. The trends represented by all previous sets hold true for this one as well. MPI is faster and it exhibits better scalability due to its pairwise communication. The MPI code is between 2.38 and 4.1 times as fast, exhibiting scaling of 1.68x, 1.67x, and 1.46x as processes double. The Titanium code has scaling of 1.61x, 1.34x, and 1.11x as processes double. The MPI code takes approximately 4.5 times as long to compute, and the Titanium code takes approximately 4 times as long, which are both consistent with the O(n^2) complexity.

Figure 26: DM Transform - 2048x2048 Matrix



DM Transform - 2048x2048 Matrix Figure 26 shows the results for the $2048 * 2048$ matrix. These results are similar to those in the smaller matrix runs. Titanium had scaling of 1.6x, 1.35x, and 1.08x as processes doubled. The MPI code had scaling of 1.68x, 1.73x, and 1.45x as processes doubled. The MPI code was 2.37x to 4.1x times as fast as the Titanium code. Both the Titanium and MPI code maintained the $O(n^2)$ complexity, each taking 4 times as long to solve as the 1024x1024 matrix.

DM Transform Notes The transform code behaved much differently on distributed memory than shared memory. One interesting thing to note is that the MPI code consistently scales better than the Titanium code, especially when processes jump from 8 to 16. The Titanium code scales very poorly at this size. This is because the global array in Titanium resides solely in a single memory region. All processes that are not in this region (half the processes, which reside remotely on the other node) require remote references over the network to get and put data in the matrix. This slows down the scaling, as half the cores are attempting to access a single region of memory.

The MPI code uses pairwise communication between decomposed submatrices. The back and forth communication scales consistently as processes are doubled. Therefore, MPI code as a whole scales better than the Titanium code.

One solution to this Titanium problem is using a truly distributed array that is still global [31].

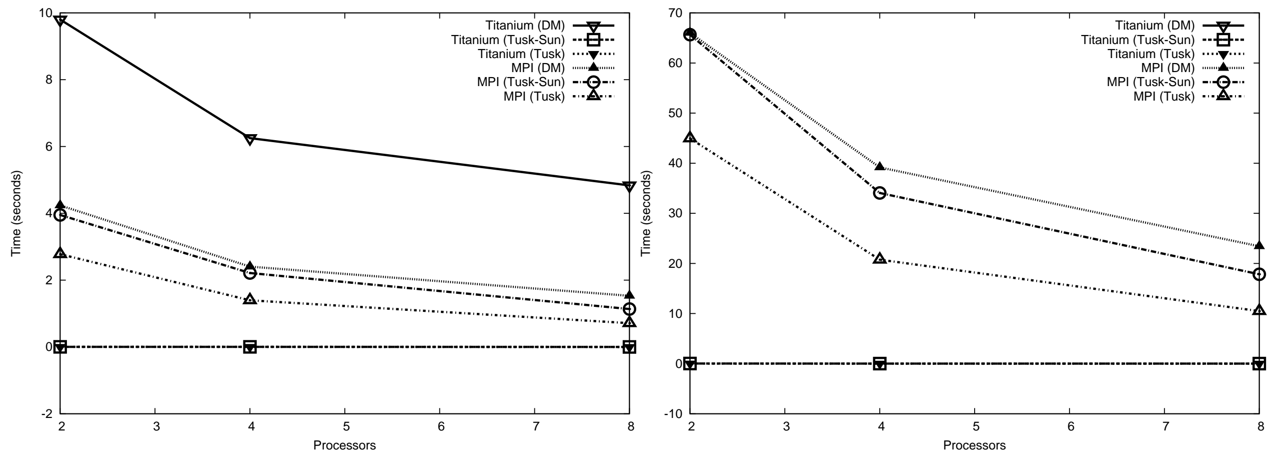
A matrix defined as thus would be distributed between regions at startup, but still viewed as local. The Titanium code was not optimized to do this, however, and worse scaling is the result.

4.2.6 Tusk/Tusk-Sun Transform Results

This section is similar to the previous comparison between the Tusk and Tusk-Sun Multiply (4.2.3). In these graphs, there are six plots. Three are for Titanium, and three are for MPI. Each code is run on distributed memory, shared memory on Tusk, and shared memory on Tusk-Sun. They are notated as T-Ti/MPI, TS-Ti/MPI, and DM-Ti/MPI.

Figure 27: Tusk/Tusk-Sun Comparison Runs - Transform of length 256x256

Figure 28: Tusk/Tusk-Sun Comparison Runs - Transform of length 1024x1024



Tusk/Tusk-Sun Transform 256x256 Figure 27 shows a number of different runs. The two Titanium codes that run on Tusk and Tusk-Sun in shared memory are both extremely fast due to use of local loads and stores. The number of total accesses made is so small that the runtimes between the T-Ti and TS-Ti codes are negligible. On distributed memory, the DM-Ti code is much slower than either shared memory code, taking approximately 10 seconds to compute the matrix transform, compared to less than .003 seconds for the shared memory code.

The T-MPI, TS-MPI, and DM-MPI code all exhibit similar scaling. The performance jumps observed, where T-MPI is faster than TS-MPI, which is faster than DM-MPI, was first observed

on the earlier comparison of multiply comparisons between Tusk and Tusk-Sun. See Figures 15 and 16 for those runtimes. These differences happen because the T-MPI code is processed the fastest. The TS-MPI code requires at least the lowest common processor speed due to a **gather** call collecting data at the end of program completion, and then requires extra speed for remote message passing over the network, as opposed to across a single memory.

Tusk/Tusk-Sun Transform 1024x1024 Figure 28 shows similar scaling and performance trends to the transform with matrix size=$256 * 256$. The DM-Ti code is not seen in this graph- with 8 processors its runtime is 72 seconds. The MPI code still exhibits the similar scaling as seen in Figures 27, 15, and 16, with distributed memory being the slowest.

4.2.7 Performance Results Notes

The performance results across differing platforms, matrix sizes, number of cores, and on shared and distributed memory demonstrate some important computational differences between the Titanium and MPI code.

First, the programming model of Titanium is a double-edged sword. The ability to use GAS is an advantage on shared memory runs. A "naive" GAS abstracted Titanium multiply kernel is shown to perform better on shared memory than an optimized "real" application- one that would normally be written following a parallel decomposition model. Additionally, by using the GAS for a matrix transform in shared memory, communication is shown to consistently be almost 1000 times as fast as equivalent MPI code. The Titanium code also shows better scaling for shared memory jobs, getting near 2 times speedup as processes double.

Despite these observations, the Titanium code usually performs more slowly than the MPI though, for shared memory multiply jobs, except in the $2048 * 2048$ multiply using 16 processes. This is in our opinion a compiler problem, one that could be fixed with a faster C compiler, as opposed to the gcc compiler used by the Titanium compiler. Further research would be required to validate or refute this.

When the Titanium and MPI code is compiled and run on distributed memory, the naive Titanium code immediately shows problems with computation efficiency. The problem of the global address space is exposed, and the programming style is exposed as flawed for distributed implementation. One set of distributed runs is done with the naive code. This is sufficient to demonstrate the inefficiency of this model.

By slightly modifying the code to do data copying, the Titanium multiply performs at a comparable runtime level to the MPI code. The distributed runs still slower than the MPI code, but it exhibits the expected scalability of a parallel application.

Once again, this slow performance is hypothesized to be due to the use of unoptimized backend compiler work. The MPI backend was used instead of the GASNet layer recommended by Titanium documentation and research. Using a different backend could improve parallel performance significantly for all distributed runs, but this must be studied in future work.

For distributed memory transform runs, the Titanium code exhibits worse scaling as the number of processes increases. At 16 cores, scalability is very low. The MPI code scales much better, due to its pairwise communication scheme. The Titanium code is slowed by network contention with all remote processes accessing a single region of memory. By distributing the array among memory regions, it is expected that this problem will be resolved, and the Titanium code will have better scalability.

4.3 Productivity Results

This section presents results from programmability testing done on the programs. We first show the lines of code (LoC), number of characters (NoC), and characters per line (CpL) for each application. Then the efficiency equations are applied to get the sequential-to-parallel conversion effort of the MPI and Titanium code. The parallel conceptual complexity is scored for each language, and notes on code development time are given. At the end of this section the results of each benchmark are summed.

4.3.1 Lines of Code and Number of Characters

The lines of code, number of characters, and average number of characters per line are presented here. In both programs, MPI has the most LoC and NoC, by a wide margin. Titanium and Fortress both use fewer lines and characters, even though Titanium has relatively high CpL. One reason that Titanium has Java as a base language is because Java is a compact language [31], and its conciseness is illustrated against C code here.

Figure 29: LoC Multiply

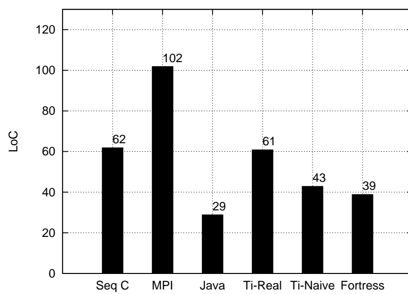


Figure 30: NoC Multiply

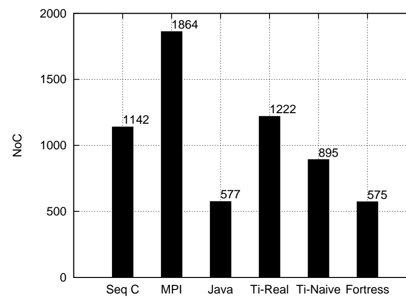
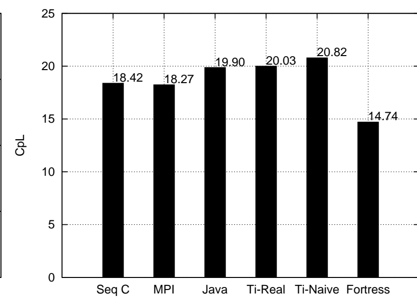


Figure 31: CpL Multiply



Figures 29, 30, and 31 show LoC, NoC, and CpL results for the multiply code. The Titanium code has less LoC than the sequential C code, and much less than the MPI code. The Titanium code LoC is double that of the sequential Java code. Both parallel codes require a good deal of extra lines, with MPI code using 40 additional lines of code, and Ti-Real code an additional 32 LoC. The Ti-Naive code only added 14 LoC from the sequential Java code, and exhibited similar performance to the Ti-Real on shared memory testing.

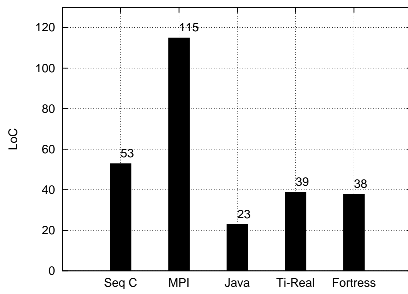
Figures 32, 33, and 34 are the tables for the Transform code. The Titanium code required 16 additional LoC, and used 317 NoC. The MPI code more than doubles the sequential C LoC, and uses an additional 742 NoC.

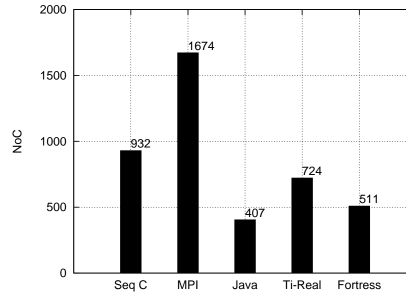
The MPI code in both programs exhibited a high increase in LoC and NoC. The CpL actually decreased in both programs, however. This is because of the high number of **for** loops used in coding. Since the coding standards require brackets to reside on their own line, each for loop uses 2 brackets that each reside on their own line. This increases LoC, and lowers CpL as a line only

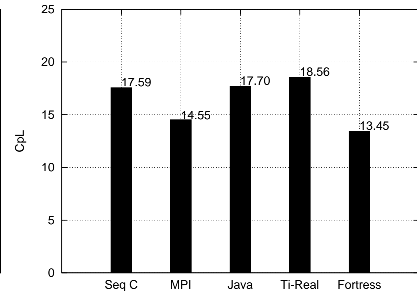
Figure 32: LoC Transform

Figure 33: NoC Transform

Figure 34: CpL Transform







has 1 character on it. The same standard applies to Titanium, but through use of **foreach** calls, Titanium cuts down on the number of stated loops, and uses less 1 character, bracket lines. In fact, the use of less **for** loops actually helps make the Titanium code have a higher CpL than the Java code in all programs.

Fortress overall has the lowest NoC. As Fortress desires to emulate mathematical notation, it has concisely expressed syntax. NoC was lowered through use of semantics like concise for loops and being able to print by simply using a **print** command, as opposed to **System.out.print** that Java and Titanium use. The semantics for **for** loops are also more concise than those of C or Java/Titanium.

4.3.2 Sequential-to-Parallel Conversion Effort

Sequential-to-parallel conversion displays the effort required to code from a sequential base into parallel. This is communicated through two equations that measure the difference in LoC and NoC between the sequential and parallel versions of code. The evaluated conversion efforts are shown in Table 2.

Table 2: Percent Conversion Efficiencies

	Multiply			Transform	
	MPI	Ti naive	Ti-Real	MPI	Ti
LoC % Effort	64.5%	48.28%	110.34%	116.98%	43.48%
NoC % Effort	63.22%	55.11%	111.79%	79.61%	66.89%

The results from Figure 2 show that MPI's transform code requires the most effort into writing new lines of code. The Titanium multiply code requires the most effort in NoC. This makes sense as Titanium utilizes some clever communicationless bounds checking to perform copy operations between processors. This overhead code is the biggest cause of the high percentages for Titanium. See lines 42-53 in Appendix A, Titanium Matrix Multiply, to see the specific code.

Although less LoC were added to Titanium code in both multiply and transform code from Java, the effort percentage is still higher. This is because the base Java code is half of what the base C code is.

4.3.3 Parallel Conceptual Complexity

In this section, we apply the complexity metrics defined in section 3.2 to the benchmark code, and develop a "score" for each benchmark and language. This benchmark measures the complexity of certain constructs by evaluating their parameters. The tables below break down the parallel conceptual complexity of the codes and give scores for each code. The tables are divided into Work Distributors (WD), Data Distributors (DD), Communicators, Synchronization and Consistency (SC) calls, and other miscellaneous calls.

Table 3: Titanium Naive Matrix Multiply Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	13		6			19	40
Calls	6		3	1	2	12	
R/S	4		3		2	9	
Notes : Score comes from 2 if, 1 foreach, 3 for, 3 broadcast, 1 barrier, 1 Ti.thisProc, and 1 Ti.numProcs statements.							

The basic, "naive" Titanium code has a score of 40, as shown in Table 3. By slightly altering the code to distribute and collect data, the score goes up 48, as shown in Table 4. The modified code uses 3 copy operations to copy data from 2 global arrays into local arrays, and then copy a local array back to part of a global array. This only requires 6 extra parameters, 2 for each copy operation, and keeps the complexity scores very similar.

Table 4: Titanium Real Matrix Multiply Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	9	3	12			24	48
Calls	5	3	6	1	2	17	
R/S	2		3		2	7	
Notes : Score comes from 2 if, 2 foreach, 1 for, 3 local, 3 broadcast, 3 copy, 1 barrier, 1 Ti.thisProc, and 1 Ti.numProcs statements.							

Table 5: Fortress Matrix Multiply Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	15					15	21
Calls	5			1		6	
R/S							
Notes : Score comes 5 for loops and 1 atomic..do statement.							

Table 5 shows that Fortress has a score of 21. The Fortress code only uses a set of 3 nested for loops to perform its matrix multiply (the other 2 are for populating the array). An **atomic..do** statement is used to ensure modification consistency when a data point is modified. This keeps Fortress's parallel conceptual complexity extremely low. Besides **the atomic..do** statement, this code looks almost exactly like sequential code. This is very important for Fortress- the languages wants parallelism implicit and sequentialism must be explicitly requested. Therefore its parallel complexity must be low. This is achieved in this code.

Table 6: MPI Matrix Multiply Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	24	7	21	1	6	59	91
Calls	10	5	3	1	5	24	
R/S	3		3		2	8	
Notes : Score comes from 3 if, 7 for, 2 malloc, 1 memset, 2 free, 1 MPI_Scatter, 1 MPI_Bcast, 1 MPI_Gather, 1 MPI_Barrier, 1 include "mpi.h", 1 MPI_Init, 1 MPI_Comm_rank, 1 MPI_Comm_size, and 1 MPI_Finalize statement.							

The MPI multiply code has a high complexity score- 91, almost double the score for Ti-Real

code, and triples the Ti-Naive score. Even when the Ti-Real code does array copying similar to the MPI code, the gap is still large. This means that coding parallel operations to do a matrix multiply in MPI is much more complex than in Titanium.

Table 7: Titanium Matrix Transform Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	10		3			13	29
Calls	5		1	1	2	9	
R/S	4		1		2	7	
Notes : Score comes 2 if, 1 foreach, 2 for, 1 broadcast, 1 barrier, 1 Ti.thisProc, and 1 Ti.numProcs statement.							

The Titanium transform code is 29. The two notable parallel calls here are a **broadcast** and **barrier** call, used to point processes to the global array, and to ensure that all computation has finished at the end of the code. The transform code then uses the GAS provided to do implicit communication. This keeps its score low.

Table 8: Fortress Matrix Transform Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	12					12	17
Calls	4			1		5	
R/S							
Notes : Score comes 4 for loops and 1 atomic..do statement.							

Table 8 shows the complexity score for the Fortress transform code is 17. The transform code is very similar to the Fortress multiply code, requiring 1 less **for** loop than the multiply code. It therefore has a similar score to the multiply code.

The MPI code has a score of 113 for the more communication intensive matrix transform code. The Titanium score is barely more than 1/4 of the MPI transform score.

Part of this is due to the heavy use of **for** loops in the MPI code. Titanium's built-in **foreach** loop that eases bounds checking. Instead of nesting **for** loops, which are expensive in LoC, NoC,

Table 9: MPI Matrix Transform Parallel Complexity

	WD	DD	Comm	SC	Misc	Sub Totals	Score
Params	23	7	42		6	78	
Calls	9	5	6		5	25	113
R/S	2		6		2	10	
Notes : Score comes 2 if, 7 for, 2 malloc, 1 memset, 2 free, 1 MPI_Scatter, 2 MPI_Send, 2 MPI_Recv, 1 MPI_Gather, 1 include "mpi.h", 1 MPI_Init, 1 MPI_Comm_rank, 1 MPI_Comm_size, and 1 MPI_Finalize statement.							

and parameters, a **foreach** makes boundary checking handled by the compiler, and makes code shorter and easier to read.

Another advantage for Titanium is that its function calls usually use less variables than similar MPI code. MPI communicators used in these programs have between five and eight parameters in every construct. The most that Titanium has in a communicator is two.

One more thing to note is that by when the naive Titanium code is modified to do data copying, it closely resembles the MPI code's layout and structure. This is meant in the way that it explicitly partitions and scatters the *A* matrix among processes, so each one has a smaller *Alocal* matrix. In this sense, the Titanium code strays completely away from its GAS abilities, and behaves in the same vein as a message passing language by copying parts of data structures, like the MPI code.

4.4 Code Development Time

As stated in the Methodology section, no strict timers were kept to record code development time. However, this section will lay out three debugging problems that each required a considerable amount of time to overcome. Each problem led to multiple hours of debugging, and required outside assistance for help in a solution. There was one notable problem for each language.

4.4.1 Implementations Problems

This section will discuss some notable problems that each required a considerable amount of time to overcome. Each problem led to multiple hours of debugging, and required outside assistance for

help in a solution. There was one notable problem for each language.

MPI : The **MPI_Scatter** call required numerous hours of debugging, profiling, and testing. The problem encountered with **MPI_Scatter** was that it only allocates a contiguous chunk of memory, and the C array was not allocated contiguously. This required a complete rewrite of the array allocation method, after other debugging attempts had failed.

Titanium : The **broadcast A from x** call in Titanium takes the pointer A in every process and points it toward the A located in process x. This is a common source of performance faults because all references to A are remote. If A is a pointer to an array located in process x's region, then all array accesses are remote calls as well. This is the problem that the naive Titanium code has when running on distributed memory. This is a difficult to debug problem because it is still a semantically correct way to program, and still maps well to shared memory. There has been Titanium based research into unintentionally referencing global pointers [99, 67].

Fortress : The biggest problem with Fortress right now is that only a tiny core of the language is implemented, and a large portion of the language specification does not work. This is said to have changed with the release of Fortress version 1.0 on April 1st, 2008. The lack of available support and reference material for Fortress currently in the community makes for learning a language especially difficult. One particular problem that this work encountered with Fortress is its current inability to cast strings to integers from a command line argument. This is hopefully corrected in the new release.

4.4.2 Productivity Notes

Titanium and Fortress generally have much better scores in complexity than MPI. The LoC and NoC for these languages is also much lower than MPI. Titanium CpL is generally high, and it has a high score for converting the Java multiply code to a "real" Titanium application. The low scores in parallel complexity heavily favor Titanium over MPI, though. All three programs experienced

some sort of implementation difficulty that requires a workaround, so it can be concluded that development in all three languages can still lead to required debugging. The overall parallel complexity scores, as well as the far lower LoC and NoC, all show that Titanium is an easier parallel model in which to program.

4.5 Holistic Evaluation

Reviewing the performance trends here, it is notable that both MPI and Titanium can scale in widely different ways depending on implementation and reliance on GAS programming and global variables. Titanium code that relies on GAS in shared memory is actually optimized code, but in distributed memory this code is hurt by the global references.

For distributed matrices, Titanium usually exhibits better scalability for the multiply code, and MPI for the transform code. However, for all distributed runs, MPI is shown to give better performance using this system setup. Therefore, the performance edge is given to MPI, as it usually has faster runtimes, especially for distributed memory.

For productivity measurements such as LoC, conversion effort, and conceptual complexity, Titanium is consistently better than MPI (except for the conversion efficiency of the refined Titanium multiply code). This is especially notable in the parallel conceptual complexity metrics, where MPI is 2-3 times as complex as Titanium. Many people have said that MPI is a complex way to perform parallel computation, and these benchmarks quantify these statements. Therefore, Titanium is determined to be more usable and programmable than MPI, as it receives generally better productivity scores.

It should be noted that Fortress was not evaluated to a final score. It is merely presented in some productivity results. Because Fortress is not a complete standard yet, it is impossible to quantify productivity scores on it. However, implementing some basic metrics on Fortress show that even with only a small subset of the language working, good programmability looks promising. Future work on the language should provide more concrete analysis of both performance and productivity abilities of the language, and provide a better holistic characterization of the language.

5 Conclusions and Future Work

This paper performs research on three parallel programming models, two languages and one standard. It gives a history of HPC and parallel computing, and gives a state of the description on MPI, Titanium, and Fortress.

Titanium and Fortress are parallel languages, and MPI is the message passing standard for HPC. Titanium and MPI use an SPMD style of parallelism by running statically created threads in parallel and communicating by sharing or copying data. Fortress is designed to be implicitly parallel. Currently this is shown through **for** loops and **tuple** operations.

Titanium threads can communicate through a global address space, partitioned among processes, or may copy parts of data structures between memory regions. By using a GAS, Titanium has a shared-memory illusion of programming, where a global variable may be seen by all processes. Titanium is a PGAS language though. Each memory region has local memories as opposed to the global one. Data may be copied or located in local memory. This can have performance advantages in distributed memory.

Two important HPC problems were coded, a matrix multiply and a matrix transform. The multiply stresses computation, and the transform communication.

It is shown that programming a "naive" Titanium multiply kernel using only GAS programming is faster than a refined "real" implementation of the code on a shared memory machine. In distributed memory, using "naive" GAS-only programming leads to intense performance degradation. For large and complex programs that implement both shared data structures and copying, special care must be taken to ensure that GAS references to variables are not done at such a rate that they impede performance in the way that the multiply kernel did. Debugging global variables can be extremely difficult, and further research has been done on this [99], because a semantically correct program with undesired global accesses is difficult to spot. By using the "real" Titanium kernel, the multiply kernel performs well and shows reasonable scaling on distributed memory.

The Titanium transform kernel exhibits less scalability than the MPI transform kernel as cores increase in distributed memory. This is because the MPI code communicates in a pair-wise way

over the network, and the Titanium code communicates with the data structure that was in only one process region. The same code in shared memory averages about 1000 times the speed of the MPI code, due to its compiler optimization. Global accesses are changed to local ones.

Titanium is usually slower than MPI in most runs, except for the shared memory transforms. It is likely that a better compiler build will increase Titanium performance, but further work needs to be done. It is possible that using different build options with the Titanium compiler, such as the Intel C compiler, or using the GASNet messaging layer, can bring Titanium's runtimes closer to or even below MPI's. It has been shown that one-sided communication paradigms are faster than two-way MPI [66], but to do so, they must make use of "smart" data partitioning and locality, as opposed to a "naive," purely GAS-based approach.

It is also reasonable to scale the programs to larger problems sets to further test scalability. Titanium's distributed memory scalability is close to MPI's, with both programs exhibiting slightly less scalability as the number of processes increases from 8 to 16 cores. Tusk and Tusk-Sun only comprise 16 cores. We do not scale beyond that.

In regards to productivity testing, it is shown that Titanium is more programmable and productive than MPI, and that Fortress shows great promise as a productivity language. The current state of Fortress cripples full productivity and performance testing. Future work can glean more about these characteristics.

6 Appendix A : Program Code

6.1 Fortress Code for Matrix Multiplication

```
1 component mm
2   export Executable
3
4 make_matrix(m:ZZ32, n:ZZ32): Array[\RR64, (ZZ32,ZZ32) \] =
5  array[\RR64\](m,n)
6
7 (*
8 print_matrix(rows:ZZ32, cols:ZZ32, mat:Array[\RR64, (ZZ32,ZZ32)\]):() = do
9   println "printing"
10   for i <- seq(0#rows)
11   do
12     for j <- seq(0#cols)
13     do
14       print mat[i,j] " "
15     end
16     println " "
17   end
18 end
19 *)
20
21 run(args:String...):()=do
22   N :ZZ32 = 256
23   tt:RR64 := (t2 - t1) / 10^6
24
25   println "Creating matrices with sides of length " N
26   A = make_matrix(N,N) (* create matrices *)
27   B = make_matrix(N,N)
28   C = make_matrix(N,N)
29
30   for i <- 0#N-1
31   do                             (* Populate matrices *)
32     for j <- 0#N-1
33     do
34       A[i,j] := i+j
35       B[i,j] := i+j
36       C[i,j] := 0
37     end
38   end
39
40   println "Doing matrix multiply" (* Start timing *)
41   t1 := nanoTime()
42   for i<- 0#N-1  (* Do matrix multiplication *)
43   do
44     for j <- 0#N-1
45     do
46       for k <- 0#N-1
47       do
48         atomic
49         do
50           C[i,j] := C[i,j] + A[i,k] B[k,j]
51         end
52       end
53     end
54   end
55   t2 := nanoTime()  (* Stop timing *)
56   tt := (t2 - t1) / 10^6
57   println "Time to complete matrix multiplication = " (tt/1000.0) " seconds"
58   end
59 end
```

6.2 Fortress Code for Matrix Transform

```
1 component transform
2 export Executable
3
4 make_matrix(m:ZZ32, n:ZZ32): Array[\RR64, (ZZ32,ZZ32) \] =
5 array[\RR64\](m,n)
6
7 (*
8 print_matrix(rows:ZZ32, cols:ZZ32, mat:Array[\RR64, (ZZ32,ZZ32)\]):() = do
9   println "printing"
10   for i <- seq(0#rows) do
11     for j <- seq(0#cols) do
12       print mat[i,j] " "
13     end
14     println " "
15   end
16 end
17 *)
18
19 run(args:String...):()=do
20
21   N :ZZ32 = 512
22
23   println "Creating matrix with sides of length " N
24    A = make_matrix(N,N)
25
26   for i <- 0#N-1
27     do                            (* Populate matrices *)
28     for j <- 0#N-1
29       do
30       A[i,j] := (N i)+j
31     end
32   end
33   println "Doing matrix transform"
34   t1 := nanoTime()
35   for i<- 0#N-1
36     do                              (* Do rotate *)
37     for j <- 0#N-1
38       do
39       atomic
40  do
41         temp:ZZ32:=A[i,j]
42  A[N-1-i,N-j-1] = A[i,j]
43  A[i,j] = temp
44      end
45    end
46   end
47   t2 := nanoTime()
48   tt := (t2 - t1) / 10^6
49
50   println "Time to complete matrix multiplication = " (tt/1000.0) " seconds"
51
52 end
53 end
```

6.3 Sequential Java Code for Matrix Multiplication

```
1 public class mm
2 {
3   public static void main(String[] args)
4    {
5  // variables
6     int N = 512; // matrices are size NxN
7     double[][] A; // instantiate matrices
8     double[][] B;
```

```
9     double[][] C;
10
11     System.out.println("Creating matrices of size "+N);
12     A = new double[N][N];
13     B = new double[N][N];
14     C = new double[N][N];
15
16     for(int i=0;i<N;i++)
17       for(int j=0;j<N;j++)
18       {
19  A[i][j]=(i+j); // populate A
20         B[i][j]=(i+j); // populate B
21  C[i][j]=0.0; // zero C
22       }
23
24     System.out.println("Doing matrix multiply.");
25     double t1 = System.currentTimeMillis(); // start timing
26     for(int i=0;i<N;i++) // do matrix multiply
27       for(int k=0;k<N;k++)
28         for(int j=0;j<N;j++)
29    C[i][j] += A[i][k]*B[k][j];
30          double t2 = System.currentTimeMillis(); // stop timing
31
32     System.out.println("Time to complete matrix multiplication = "+(t2-t1)/1000);
33   }
34 }
```

6.4 Sequential Java Code for Matrix Transform

1 class transform 2 3 public static void main(String[] args) 4 5 // variables 6 int N; 7 double[][] a; 8 N = 512; 9 a = new double[N][N]; // our matrix 10 11 for(int i=0;i¡N;i++) 12 for(int j=0;j¡N;j++) 13 a[i][j]=i*N+j; // populate A 14 15 double t1 = System.currentTimeMillis(); // start timing 16 for(int i=0;i¡N/2;i++) 17 for(int j=0;j¡N;j++) 18 19 double temp = a[i][j]; // switch 2 points 20 a[i][j] = a[N-1-i][N-1-j]; 21 a[N-1-i][N-1-j] = temp; 22 23 double t2 = System.currentTimeMillis(); // stop timing 24 /* 25 for(int i=0;i¡N;i++) 26 27 for(int j=0;j¡N;j++) 28 System.out.print(a[i][j]+" "); 29 System.out.println(""); 30 31 */ 32 System.out.println("Time to complete matrix rotate = "+(t2-t1)/1000); 33 34

6.5 Sequential C Code for Matrix Multiply

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <time.h>
5 //void print_matrix(int, int, double**);
6 double **allocate_matrix(int,int);
7 void deallocate_matrix(double **array, int row_dim);
8
9 main(int argc, char *argv[])
10 {
11  // variables
12   double ** A; // our 3 matrices
13   double ** B;
14   double ** C;
15   int N = 512; // matrices are NxN
16   int i; // iterator values
17   int j;
18   int k;
19   clock_t t1; // timing variables
20   clock_t t2;
21   float ratio = 1./CLOCKS_PER_SEC;
22
23   printf("Creating matrices of size %d\n",N);
24   A = allocate_matrix(N,N); // allocate matrices
25   B = allocate_matrix(N,N);
```

```
26   C = allocate_matrix(N,N);
27   for(i = 0; i < N; i++)
28     for(j = 0; j < N; j++)
29     {
30       A[i][j] = (i+j); // populate A
31       B[i][j] = (i+j); // populate B
32       C[i][j] = 0.0; // zero C
33     }
34
35   printf("Doing matrix multiply\n");
36   t1 = clock(); // start timing
37   for(i = 0; i < N; i++) // do matrix multiply
38     for(j = 0; j < N; j++)
39       for(k = 0; k < N; k++)
40         C[i][j] += A[i][k] * B[k][j];
41   t2 = clock(); // stop timing
42
43   printf("Time to complete matrix multiplication
 = %f seconds\n",ratio*(long)t1 + ratio*(long)t2);
44
45   deallocate_matrix(A,N); // deallocate matrices
46   deallocate_matrix(B,N);
47   deallocate_matrix(C,N);
48 }
49  // common C methods
50 double **allocate_matrix(int rows, int cols)
51 {
52   int i;
53   double **mat;
54   double *mat2;
55
56   mat2= (double *)malloc(rows*cols * sizeof(double));
57   memset(mat2,0,rows*cols*sizeof(double));
58   mat=(double **)malloc(rows*sizeof(double *));
59   for(i=0;i<rows;i++){
60     mat[i]=&(mat2[i*cols]);
61   }
62   return mat;
63 }
64
65 void deallocate_matrix(double **array, int row_dim)
66 {
67   int i;
68   for(i=1; i<row_dim; i++)
69         array[i]=NULL;
70   free(array[0]);
71   free(array);
72 }
73 /*
74 // pretty print a matrix
75 void print_matrix(int rows, int cols, double **mat)
76 {
77   int i;
78   int j;
79
80   for(i=0;i<rows;i++)
81   {
82     printf("%g",mat[i][0]);
83     for(j=1;j<cols;j++)
84     {
85       printf(",%g ",mat[i][j]);
86     }
87     printf("\n");
88   }
89 }
90 */
91
92
```

6.6 Sequential C Code for Matrix Transform

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 //void print_matrix(int, int, double**);
7 double **allocate_matrix(int,int);
8 void deallocate_matrix(double **, int);
9
10 main(int argc, char * argv[])
11 {
12  // variables
13   int N = 8; // matrix is NxN size
14   double ** a;
15   int i; // iterator values
16   int j;
17   double temp;
18   clock_t t1;
19   clock_t t2;
20   float ratio = 1.0/CLOCKS_PER_SEC;
21
22   a = allocate_matrix(N,N); // allocate matrix
23
24   for(i = 0; i < N; i++) // populate matrix
25     for(j = 0; j < N; j++)
26       a[i][j] = i*N+j;
27
28   t1 = clock(); // start timing
29   for(i = 0; i < N/2; i++) // do transform
30     for(j = 0; j < N; j++)
31     {
32       temp = a[i][j];
33       a[i][j] = a[N-1-i][N-1-j];
34       a[N-1-i][N-1-j] = temp;
35     }
36   t2 = clock(); // stop timing
37
38   printf("Time to do transform = %f seconds\n",ratio*(long)t1 + ratio*(long)t2);
39   deallocate_matrix(a,N);
40 }
41
42 double **allocate_matrix(int rows, int cols)
43 {
44   int i;
45   double **mat;
46   double *mat2;
47   mat2= (double *)malloc(rows*cols * sizeof(double));
48   memset(mat2,0,rows*cols*sizeof(double));
49   mat=(double **)malloc(rows*sizeof(double *));
50   for(i=0;i<rows;i++)
51   {
52     mat[i]=&(mat2[i*cols]);
53   }
54   return mat;
55 }
56
57 void deallocate_matrix(double **array, int row_dim)
58 {
59   int i;
60   for(i=1; i<row_dim; i++)
61         array[i]=NULL;
62   free(array[0]);
63   free(array);
64 }
65
66 /*
```

```
67 void print_matrix(int rows, int cols, double **mat)
68 {
69   int i;
70   int j;
71
72   for(i=0;i<rows;i++)
73   {
74     printf("%g",mat[i][0]);
75     for(j=1;j<cols;j++)
76     {
77       printf(",%g ",mat[i][j]);
78     }
79     printf("\n");
80   }
81 }
82 */
```

6.7 "Naive" Titanium Code (Ti-Naive) for Matrix Multiply

```
1 public class mm_naive
2 {
3   public static void main(String[] args)
4    {
5  // variables
6     int single N; // matrices are NxN
7     int rank = Ti.thisProc(); // titanium variables
8     int size = Ti.numProcs();
9     Timer t = new Timer();
10     N = 512; // our matrix is size NxN
11     int interval = N / size; // number of rows of Alocal and Clocal
12
13  // create multidimensional arrays
14     Point<2> l = [0, 0];
15     Point<2> h = [N, N];
16     RectDomain<2> r = [ l : h ];
17     double [2d] A = new double[r]; // matrices default to global
18     double [2d] B = new double[r];
19     double [2d] C = new double[r];
20
21     if(rank==0)    // root process populates matrices
22     {
23       System.out.println("Creating matrices with sides of length = "+N);
24       System.out.println(" Interval for each process = "+interval);
25       foreach(p in A.domain())
26       { // populate A
27     A[p] = p[1] + p[2]; // populate A
28    B[p]=p[1]+p[2]; // populate B
29      C[p]=0; // zero C
30       }
31       System.out.println("Doing parallel matrix multiply.");
32       t.start();
33     }
34
35     B = broadcast B from 0; // super inefficient... every
36     A = broadcast A from 0; // reference is a remote read..
37     C = broadcast C from 0; // not good for distributed memory
38
39     for(int i=interval*rank;i<interval*rank+interval;i++) // do multiplication
40       for(int k = 0; k < N;k++)
41  for(int j=0;j<N;j++)
42    C[i,j] += A[i,k]*B[k,j];
43
44     Ti.barrier(); // barrier ensures completion
45     if(rank==0)
46     {
47       t.stop();
```

```
48       System.out.println("Time to complete matrix multiplication
 and communication = "+t.secs()+ " seconds");;
49     }
50   }
51 }
```

6.8 "Real" Titanium Code (Ti-Real) for Matrix Multiplication

```
1 public class mm
2 {
3   public static void main(String[] args)
4   {
5  // variables
6     int N; // matrices are NxN
7     int rank = Ti.thisProc(); // titanium variables
8     int size = Ti.numProcs();
9     Timer t = new Timer(); // timer object
10     N = 512; // our matrix is size NxN
11     int interval = N / size; // number of rows of Alocal and Clocal
12
13  // create multidimensional arrays
14     Point<2> l = [0, 0];
15     Point<2> h = [N, N];
16     RectDomain<2> r = [ l : h ];
17     double [2d] A; // matrices default to global
18     double [2d] B;
19     double [2d] C;
20
21     if(rank==0)    // root process populates matrices
22     {
23       System.out.println("Creating matrices with sides of length = "+N);
24       System.out.println("  and local matrices with lengths of rows =  "+interval);
25       A = new double[r]; // instantiate matrices
26       B = new double[r];
27       C = new double[r];
28       foreach(p in A.domain())
29       {
30     A[p] = p[1] + p[2]; // populate A
31    B[p]=(p[1]+p[2]); // populate B
32      C[p]=0; // zero C
33       }
34       System.out.println("Distributing matrices and doing matrix multiply");
35       t.start(); // start timing
36     }
37     Ti.barrier();
38     A = broadcast A from 0; // doing this gives each process
39     B = broadcast B from 0; // a pointer to A,B,and C,so we can
40     C = broadcast C from 0; // do a copy operation
41
42     int startIndex = rank * interval; // local bounds handling,
43     int endIndex = startIndex+interval-1; // each process copies a unique
44     Point<2> startPoint = [ startIndex, 0 ]; // portion of A to its Alocal
45     Point<2> endPoint = [ endIndex, N ];
46     RectDomain<2> myPart = [startPoint:endPoint];
47
48     double [2d] local Alocal = new double[myPart]; // create local arrays
49     double [2d] local Blocal = new double[r];
50     double [2d] local Clocal = new double[myPart];
51
52     Alocal.copy(A.restrict(myPart)); // copy A to Alocal matrices
53     Blocal.copy(B); // copy B to Blocal matrices
54
55     foreach(p in Alocal.domain()) // do local matrix multiplication
56     {
57       Clocal[p] = 0;
58       for(int k=0;k<N;k++)
```

```
59  Clocal[p] +=  Alocal[p[1],k]*Blocal[k,p[1]];
60      }
61
62      C.copy(Clocal); // copy Clocals to C (in proc 0)
63      Ti.barrier(); // barrier ensures all processes finish
64      if(rank==0)
65      {
66        t.stop(); // stop timing
67        System.out.println("Time to complete matrix multiplication
 and communication = "+t.secs()+ " seconds");
68      }
69    }
70 }
```

6.9 Titanium Code for Matrix Transform

```
1 public class transform
2 {
3   public static void main(String[] args)
4   {
5  // variables
6     int N; // matrices are NxN
7     int rank = Ti.thisProc(); // titanium variables
8     int size = Ti.numProcs();
9     Timer t = new Timer(); // timer object
10     N = 2048;                          // our matrix is size NxN
11     int interval = N / size;          // each process responsible for an
12                                       // interval of the matrix
13
14     Point<2> l = [0, 0]; // create matrix
15     Point<2> h = [N, N];
16     RectDomain<2> r = [ l : h ];
17     double [2d] a = new double[r];     // matrix defaults to global
18
19     if(rank==0)                        // root process populates matrix
20     {
21       System.out.println("Creating matrices with sides of length = "+N);
22       System.out.println(" Interval for each process = "+interval);
23       foreach(p in a.domain())   // populate A
24           a[p] = p[1] + p[2];
25       System.out.println("Doing matrix transform");
26       t.start(); // start timing
27     }
28
29     a = broadcast a from 0; // point all processes to the matrix
30
31     for(int i = interval*rank/2; i <interval*rank/2+interval/2;i++) // do transform
32       for(int j=0;j<N;j++)
33       {
34  double temp = a[i,j];
35  a[i,j] = a[N-1-i,N-1-j];
36  a[N-1-i,N-1-j] = temp;
37       }
38
39     Ti.barrier(); // barrier to ensure completion
40     if(rank==0)
41     {
42       t.stop(); // stop timing
43       System.out.println("Time to tranpose = ");
44       System.out.println(t.secs());
45     }
46   }
47 }
```

6.10 MPI Code for Matrix Multiplication

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <mpi.h>
5 //void print_matrix(int, int, double**);
6 double **allocate_matrix(int,int);
7 void deallocate_matrix(double **array,int row_dim);
8
9 main( int argc,char **argv )
10 {
11     // variables
12   double **A; // our 3 matrices
13   double **B;
14   double **C;
15   int N = 1024; // matrices are NxN
16   int i; // iterator variables
17   int j;
18   int k;
19   double t1; // timing variables
20   double t2;
21     // additional MPI variables
22   double **Alocal; // local matrices
23   double **Clocal;
24   int rank; // process rank
25   int size; // number of processes
26   int interval; // number of rows of Alocal and Clocal
27
28   MPI_Init(&argc,&argv); // start up MPI
29   MPI_Comm_rank(MPI_COMM_WORLD,&rank); // find each process's rank
30   MPI_Comm_size(MPI_COMM_WORLD,&size); // find number of processes
31   interval=N/size; // interval to go over
32
33   if(rank==0)
34   {
35     printf("Number of processes : %d\n",size);
36     printf("Creating matrices with sides of length %d\n",N);
37     printf("  and submatrices of size %d*%d\n",interval,N);
38   }
39
40   B=allocate_matrix(N,N);  // every process alloctes for B, Alocal, and Clocal
41   Alocal=allocate_matrix(interval,N);
42   Clocal=allocate_matrix(interval,N);
43
44   if(rank==0) { // only process 0 allocates for A and C
45     A=allocate_matrix(N,N);  // only 0 allocates matrices A and C
46     C=allocate_matrix(N,N);
47     for(i=0;i<N;i++)
48       for(j=0;j<N;j++)
49       {
50  A[i][j]=(i+j); // populate A
51  B[i][j]=(i+j); // populate B
52       }
53     printf("Distributing matrices and doing matrix multiply.\n");
54     t1=MPI_Wtime(); // start timing
55   }
56  // Scatter matrix A among processes
57  // Matrix A will be distributed into
58  // local Apart matrices of size (interval)xN
59   MPI_Scatter(A[0],interval*N,MPI_DOUBLE,Alocal[0],
interval*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
60
61  // Broadcast matrix B to everybody
62  // so every process has a local copy
63   MPI_Bcast(*B,N*N,MPI_DOUBLE, 0, MPI_COMM_WORLD);
64   for(i=0;i<interval;i++) // each process does its own local
65     for(j=0;j<N;j++) // matrix muliply
```

```
66     { // Alocal * B = Clocal
67       Clocal[i][j]=0.0;
68       for(k=0;k<N; k++)
69  Clocal[i][j]+=Alocal[i][k]*B[k][j];
70     }
71    // MPI_Gather takes the Clocal matrices and
72    // gathers them in C on process 0
73   MPI_Gather(Clocal[0],interval*N,MPI_DOUBLE,C[0],
interval*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
74   MPI_Barrier(MPI_COMM_WORLD);
75   if(rank==0)
76   {
77     t2 = MPI_Wtime();
78     printf("Time to complete matrix multiplication
and communication = %f seconds\n",t2-t1);
79     deallocate_matrix(A,N); // only process 0 deallocates A and C
80     deallocate_matrix(C,N);
81   }
82   deallocate_matrix(Clocal,interval); // everyone deallocates Alocal,
83   deallocate_matrix(Alocal,interval); // Clocal, and B
84   deallocate_matrix(B,N);
85   MPI_Finalize(); // Close down MPI environment
86 }
87
88 double **allocate_matrix(int rows, int cols)
89 {
90   int i;
91   double **mat;
92   double *mat2;
93
94   mat2= (double *)malloc(rows*cols * sizeof(double));
95   memset(mat2,0,rows*cols*sizeof(double));
96   mat=(double **)malloc(rows*sizeof(double *));
97   for(i=0;i<rows;i++){
98     mat[i]=&(mat2[i*cols]);
99   }
100   return mat;
101 }
102 void deallocate_matrix(double **array, int row_dim)
103 {
104   int i;
105   for(i=1; i<row_dim; i++)
106  array[i]=NULL;
107   free(array[0]);
108   free(array);
109 }
110 /*
111 void print_matrix(int rows, int cols, double **mat)
112 {
113   int i;
114   int j;
115   for(i=0;i<rows;i++){
116     printf("%g",mat[i][0]);
117     for(j=1;j<cols;j++){
118       printf(",%g",mat[i][j]);
119     }
120     printf("\n");
121   }
122 }
123 */
```

6.11 MPI Code for Matrix Transform

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
```

```
4 #include <mpi.h>
5
6 //void print_matrix(int,int,double **);
7 double **allocate_matrix(int,int);
8 void deallocate_matrix(double **array, int row_dim);
9
10 main(int argc, char * argv[])
11 {
12  // variables
13   int N = 2048; // matrix is NxN size
14   double **a;
15
16   int i; // iterator values
17   int j;
18   double temp; // temp value
19   double t1;
20   double t2;
21  // additional MPI variables
22   double **alocal; // local matrix
23   int rank; // process rank
24   int size; // number of processes
25   int interval; // number of rows of Alocal
26   int buddy;
27   MPI_Status status; // needed for MPI_Recv calls
28
29   MPI_Init(&argc, &argv); // start up MPI
30   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // find each process's rank
31   MPI_Comm_size(MPI_COMM_WORLD, &size); // number of processes
32   interval = N / size; // interval to go over
33   buddy = size - 1 - rank; // each process finds its buddy
34
35   alocal = allocate_matrix(interval, N); // all processes allocate for Alocal
36
37   if(rank==0)
38   {
39     a = allocate_matrix(N,N); // only process 0 allocates A
40     for(i = 0; i < N; i++)
41       for(j = 0; j < N; j++)
42         a[i][j] = i*N+j ; // populate matrix
43     t1 = MPI_Wtime(); // start timing
44   }
45  // Scatter matrix A among processes
46  // Matrix A will be distributed into
47  // local Apart matrices of size (interval)xN
48   MPI_Scatter(a[0],interval*N,MPI_DOUBLE,alocal[0],
interval*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
49
50   if(rank % 2 == 0) // even numbered process work-
51   { // send a variable and receive
52     for(i = 0; i < interval; i++) // one back
53       for(j = 0; j < N; j++)
54       {
55         MPI_Send(&alocal[i][j],1,MPI_DOUBLE,buddy,0,MPI_COMM_WORLD);
56  MPI_Recv(&alocal[i][j],1,MPI_DOUBLE,buddy,0,MPI_COMM_WORLD,&status);
57       }
58   }
59   else // rank % 2 == 1 // odd numbered process work-
60   { // receive a variable and
61     for(i = interval-1; i>=0 ; i--) // send one back
62       for(j = N-1; j>=0; j--)
63       {
64         MPI_Recv(&temp,1,MPI_DOUBLE,buddy,tag,MPI_COMM_WORLD,&status);
65         MPI_Send(&alocal[i][j],1,MPI_DOUBLE,buddy,tag,MPI_COMM_WORLD);
66         alocal[i][j] = temp;
67       }
68   }
69  // MPI_Gather takes the Alocal matrices and
70  // gathers them in A on process 0
```

```
71   MPI_Gather(alocal[0],N*interval,MPI_DOUBLE,a[0],
N*interval,MPI_DOUBLE,0, MPI_COMM_WORLD);
72
73   if(rank==0)
74   {
75     t2 = MPI_Wtime(); // stop timing
76     printf("Time for transform = %f seconds\n",t2-t1);
77     deallocate_matrix(a,N); // only process 0 deallocates A
78   }
79   deallocate_matrix(alocal,interval); // everyone deallocates Alocal
80   MPI_Finalize(); // Close down MPI environment
81 }
82
83 double **allocate_matrix(int rows, int cols)
84 {
85   int i;
86   double **mat;
87   double *mat2;
88
89   mat2= (double *)malloc(rows*cols * sizeof(double));
90   memset(mat2,0,rows*cols*sizeof(double));
91   mat=(double **)malloc(rows*sizeof(double *));
92   for(i=0;i<rows;i++){
93     mat[i]=&(mat2[i*cols]);
94   }
95   return mat;
96 }
97
98 void deallocate_matrix(double **array, int row_dim)
99 {
100   int i;
101   for(i=1; i<row_dim; i++)
102         array[i]=NULL;
103   free(array[0]);
104   free(array);
105 }
106
107 /*
108 void print_matrix(int rows, int cols, double **mat)
109 {
110   int i;
111   int j;
112
113   for(i=0;i<rows;i++){
114     printf("%g",mat[i][0]);
115     for(j=1;j<cols;j++){
116       printf(",%g",mat[i][j]);
117     }
118     printf("\n");
119   }
120 }
121 */
```

7 Appendix B : Runtime Results

This appendix presents the timing results done for the purposes of performance testing. The tables are grouped into multiplication and transform sets. Each set is broken up into shared and disributed memory (SM or DM) for either the Titanium or MPI codes. Within each table, the matrix size is

given by an "N=" notation, and the left side of the table denotes the number of processes used, either 2, 4, 8, or 16 (in DM). The corresponding right half of the table are the resulting runtimes, in seconds.

7.1 Multiplication Runtimes

7.1.1 SM MPI

N=128	
2	.01855
4	.01708
8	.01953

N=256	
2	0.13671
4	0.08593
8	0.06250

N=512	
2	1.18994
4	0.68847
8	0.44482

N=1024	
2	26.27783
4	17.07666
8	12.78466

N=2048	
2	428.61914
4	239.85498
8	180.03515

7.1.2 SM Ti-Real

N=128	
2	0.13197
4	0.06658
8	0.03404

N=256	
2	1.04984
4	0.52681
8	0.26616

N=512	
2	9.06815
4	4.55791
8	2.28854

N=1024	
2	72.10291
4	36.14697
8	18.34434

N=2048	
2	583.92459
4	292.06047
8	146.46946

7.1.3 SM Ti-Naive

N=128	
2	0.13023
4	0.06508
8	0.03287

N=256	
2	1.04116
4	0.52017
8	0.25983

N=512	
2	8.32400
4	4.16212
8	2.08140

N=1024	
2	66.83434
4	33.35030
8	16.69746

N=2048	
2	533.53498
4	266.83239
8	133.40978

7.1.4 SM Ti-Real

N=128	
2	0.13023
4	0.06508
8	0.03287

N=256	
2	1.04116
4	0.52017
8	0.25983

N=512	
2	8.32400
4	4.16212
8	2.08140

N=1024	
2	66.83434
4	33.35030
8	16.69746

N=2048	
2	533.53498
4	266.83239
8	133.40978

7.1.5 DM MPI

N=128	
2	0.04296
4	0.02246
8	0.02636
16	0.04150

N=256	
2	0.18554
4	0.10595
8	0.08300
16	0.13085

N=512	
2	4.87402
4	2.96142
8	1.78027
16	1.16064

N=1024	
2	70.3999
4	41.5986
8	23.2465
16	13.0468

7.1.6 DM Ti-Real

N=128	
2	0.25329
4	0.13098
8	0.08042
16	0.06110

N=256	
2	2.00124
4	1.02044
8	0.53530
16	0.32069

N=512	
2	16.54163
4	8.40617
8	4.35345
16	2.35468

N=1024	
2	200.99825
4	100.19784
8	51.05700
16	35.54844

7.1.7 DM Ti-Naive

N=128	
2	644.53131
4	565.15021
8	549.74546
16	504.23612

7.1.8 TS-MPI

N=256	
2	0.18017
4	0.11523
8	0.08837

N=1024	
2	70.92020
4	39.92431
8	25.04785

7.1.9 TS-Ti-Real

N=256	
2	1.01042
4	0.51061
8	0.25621

N=1024	
2	140.21077
4	70.20393
8	33.04511

7.2 Transform Runtimes

7.2.1 SM MPI

N=128	
2	0.69531
4	0.31347
8	0.20117

N=256	
2	2.77734
4	1.39564
8	0.71191

N=512	
2	10.19363
4	4.56770
8	2.81975

N=1024	
2	44.95256
4	20.74497
8	10.49023

N=2048	
2	159.02994
4	69.50846
8	41.11551

7.2.2 SM Titanium

N=128	
2	0.00068
4	0.00048
8	0.00063

N=256	
2	0.00255
4	0.00140
8	0.00107

N=512	
2	0.00837
4	0.00457
8	0.00268

N=1024	
2	0.03219
4	0.01612
8	0.00867

N=2048	
2	0.12833
4	0.06405
8	0.03417

7.2.3 DM MPI

N=128	
2	1.11523
4	0.62011
8	0.39160
16	0.30078

N=256	
2	4.23632
4	2.40429
8	1.53222
16	1.07910

N=512	
2	16.67968
4	10.10677
8	5.86979
16	3.98307

N=1024	
2	66.00651
4	39.17057
8	23.42968
16	15.98437

N=2048	
2	267.17317
4	158.35026
8	91.32682
16	62.85807

7.2.4 DM Titanium

N=128	
2	2.45000
4	1.52135
8	1.11541
16	1.01922

N=256	
2	9.80000
4	6.24960
8	4.83559
16	4.08406

N=512	
2	39.36402
4	25.19878
8	18.04365
16	16.16923

N=1024	
2	157.40089
4	97.58733
8	72.62399
16	65.53650

N=2048	
2	635.39621
4	379.23796
8	279.02332
16	257.77633

7.2.5 TS-MPI

N=256	
2	3.95117
4	2.21386
8	1.13346

N=1024	
2	65.65917
4	34.05468
8	17.83496

7.2.6 TS-Titanium

N=256	
2	0.00257
4	0.00143
8	0.00107

N=1024	
2	0.03219
4	0.01612
8	0.00867

References

[1] L. V. Kalé, "New Parallel Programming Abstractions and the Role of Compilers," in IPDPS, 2006.

[2] M. Weiland, "Chapel, Fortress and X10: Novel Languages for HPC," tech. rep., The University of Edinburgh, October 2007.

[3] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java dialect," in ACM 1998 Workshop on Java for High-Performance Network Computing, (New York, NY 10036, USA), ACM Press, 1998.

[4] E. Allen, D. Chase, J. Hallett, V. Luchango, J.-W. Maessen, S. Ryu, and G. L. S. Jr., "The Fortress Language Specification."

[5] J. Kepner and D. Koester, "HPCS Application Analysis and Application," 23 September 2003.

[6] L. Prechelt, "An Empirical Comparison of Seven Programming Languages," Computer, vol. 33, no. 10, pp. 23–29, 2000.

[7] M. Felleisen, "On the Expressive Power of Programming Languages," in ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark (N. Jones, ed.), vol. 432, pp. 134–151, New York, N.Y.: Springer-Verlag, 1990.

[8] J. Cai and R. Paige, "Towards Increased Productivity of Algorithm Implementation," SIGSOFT Softw. Eng. Notes, vol. 18, no. 5, pp. 71–78, 1993.

[9] P. J. Landin, "The next 700 programming languages," Commun. ACM, vol. 9, no. 3, pp. 157–166, 1966.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, and S. K. Weeratunga, "The NAS Parallel Benchmarks– Summary and Preliminary Results," in Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, (New York, NY, USA), pp. 158–165, ACM, 1991.

[11] J. J. Dongarra, "The LINPACK Benchmark: An Explanation," in Proceedings of the 1st International Conference on Supercomputing, (New York, NY, USA), pp. 456–474, Springer-Verlag New York, Inc., 1988.

[12] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers."

[13] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The hpc challenge (hpcc) benchmark suite," in SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, (New York, NY, USA), p. 213, ACM, 2006.

[14] B. Wilkinson and M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

[15] S. Gill, "Parallel Programming," Computer Journal, vol. 1, pp. 2–10, April.

[16] J. Holland, "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously," vol. 16, pp. 108–113, 1959.

[17] M. Conway, "A Multiprocessor System Design," vol. 4, pp. 139–146, 1963.

[18] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in Readings in Computer Architecture, (San Francisco, CA, USA), pp. 79–81, Morgan Kaufmann Publishers Inc., 2000.

[19] A. Bernstein, "Analysis of Programs for Parallel Processing," Electronic Computers, IEEE Transactions on, vol. EC-15, no. 5, pp. 757–763, Oct. 1966.

[20] M. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Comput., vol. C-21, pp. 948–960, 9 Septmber 1972.

[21] S. L. Graham, M. Snir, and e. Cynthia A. Patterson, Getting Up to Speed: The Future of Supercomputing. The National Academies Press, 2004. Report of National Research Council of the National Academies Sciences.

[22] Beowulf Cluster Computing with Linux. Cambridge, MA, USA: MIT Press, 2002.

[23] D. B. Loveman, "High Performance Fortran," IEEE Parallel and Distributed Technology, vol. 01, no. 1, pp. 25–42, 1993.

[24] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: A Machine Independent Programming Language for Parallel Computers," IEEE Transactions on Software Engineering, vol. 26, no. 3, pp. 197–211, 2000.

[25] G. E. Blelloch, "NESL: A Nested Data-Parallel Language (Version 2.6)," tech. rep., Pittsburgh, PA, USA, 1993.

[26] B. Carpenter and G. Fox, "HPJava: A Data Parallel Programming Alternative," Computing in Science and Engg., vol. 5, no. 3, pp. 60–64, 2003.

[27] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," IEEE Computational Science and Engineering, vol. 05, no. 1, pp. 46–55, 1998.

[28] W. D. Hillis and J. Guy L. Steele, "Data Parallel Algorithms," Commun. ACM, vol. 29, no. 12, pp. 1170–1183, 1986.

[29] K. Kennedy, C. Koelbel, and H. Zima, "The Rise and Fall of High Performance Fortran: An Historical Object Lesson," in

HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, (New York, NY, USA), pp. 7–1–7–22, ACM, 2007.

[30] P. Mehrotra, "Invited Lecture: Data Parallel Programming: The Promises and Limitations of High Performance Fortran," in ACPC, p. 114, 1993.

[31] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen, "Parallel Languages and Compilers: Perspective From the Titanium Experience," Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 266–290, 2007.

[32] F. Mueller, "A library implementation of POSIX threads under Unix," in Proceedings of theWinter 1993 USENIX Technical Conference and Exhibition, (San Diego, CA, USA), pp. 29–41, 1993.

[33] E. Tilevich and Y. Smaragdakis, "Portable and efficient distributed threads for Java," in Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, (New York, NY, USA), pp. 478–492, Springer-Verlag New York, Inc., 2004.

[34] M. P. I. F. MPIF, "MPI-2: Extensions to the Message-Passing Interface." Technical Report, University of Tennessee, Knoxville, 1996.

[35] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," tech. rep., Champaign, IL, USA, 1993.

[36] C. Kesselman, "High performance parallel and distributed computation in compositional CC++," SIGAPP Appl. Comput. Rev., vol. 4, no. 1, pp. 24–26, 1996.

[37] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An Introduction to the MPI Standard," tech. rep., Knoxville, TN, USA, 1995.

[38] K. Parzyszek, Generalized portable shmem library for high performance computing. PhD thesis, Ames, IA, USA, 2003. Co-Major Professor-Ricky A. Kendall and Co-Major Professor-Robyn R. Lutz.

[39] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, "Titanium Language Reference Manual," tech. rep., Berkeley, CA, USA, 2001.

[40] M. W. Mutka and M. Livny, "The Available Capacity of a Privately Owned Workstation Environment," Perform. Eval., vol. 12, no. 4, pp. 269–284, 1991.

[41] D. A. Patterson, D. E. Culler, and T. E. Anderson, "A case for NOW (networks of workstation)," in PODC '95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, (New York, NY, USA), p. 17, ACM, 1995.

[42] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," Software - Practice and Experience, vol. 23, no. 12, pp. 1305–1336, 1993.

[43] M. Baker, "Cluster Computing White Paper," 2000.

[44] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," Concurrency, Practice and Experience, vol. 2, no. 4, pp. 315–340, 1990.

[45] W. Gropp, E. Lusk, and A. Skjellum, Using MPI (2nd ed.): Portable Parallel Programming with the Message-Passing Interface. Cambridge, MA, USA: MIT Press, 1999.

[46] W. Gropp and E. Lusk, "Goals Guiding Design: PVM and MPI."

[47] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "A Message Passing Standard for MPP and Workstations," Communications of the ACM, vol. 39, no. 7, pp. 84–90, 1996.

[48] W. Gropp, "Learning from the Success of MPI," in HiPC '01: Proceedings of the 8th International Conference on High Performance Computing, (London, UK), pp. 81–94, Springer-Verlag, 2001.

[49] D. Bouillet, "Advantages Of Pthreads For Interprocess Communication."

[50] G. J. Narlikar, "Pthreads for Dynamic and Irregular Parallelism," pp. ??–??, 1998.

[51] B. Kuhn, P. Petersen, and E. O'Toole, "OpenMP Versus Threading in C/C++," Concurrency - Practice and Experience, vol. 12, no. 12, pp. 1165–1176, 2000.

[52] M. SuB, A. Podlich, and C. Leopold, "Observations on the Publicity and Usage of Parallel Programming Systems and Languages: A Survey Approach," tech. rep., University of Kassel, Wilhelmhöher Allee 73, D-34121 Kassel, Germany, 2007.

[53] H.-H. Wang, K.-C. Li, K.-J. Wang, and S.-H. Lu, "On the Design and Implementation of an Effective Prefetch Strategy for DSM Systems," The Journal of Supercomputing, vol. 37, pp. 91–112(22), July 2006.

[54] M. Chapman and G. Heiser, "Implementing Transparent Shared Memory on Clusters Using Virtual Machines," in ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 2005.

[55] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot, "OpenMosix, OpenSSI and Kerrighed: A Comparative Study," in CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2, (Washington, DC, USA), pp. 1016–1023, IEEE Computer Society, 2005.

[56] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," Computer, vol. 29, no. 2, pp. 18–28, 1996.

[57] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," tech. rep., Pittsburgh, PA, USA, 1993.

[58] M. R. Eskicioglu and T. A. Marsland, "Shared Memory Computing on SP2: JIAJIA Approach," in CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, p. 12, IBM Press, 1998.

[59] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren, "Architecture and Design of AlphaServer GS320," SIGARCH Comput. Archit. News, vol. 28, no. 5, pp. 13–24, 2000.

[60] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA Highly Scalable Server," SIGARCH Comput. Archit. News, vol. 25, no. 2, pp. 241–251, 1997.

[61] S. HZ, S. JP, O. L, and B. R, "Message Passing and Shared Address Space Parallelism on an SMP Cluster," Parallel Computing, vol. 29, pp. 167–186, 2003.

[62] E. Strohmaier, J. J. Dongarra, H. W. Meuer, and H. D. Simon, "Recent Trends in the Marketplace of High Performance Computing," Parallel Comput., vol. 31, no. 3+4, pp. 261–273, 2005.

[63] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291–312, 2007.

[64] Sun, "Sun Labs Programming Language Research Group." Fortres website.

[65] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick, "Evaluating support for global address space languages on the Cray X1," in ICS '04: Proceedings of the 18th annual international conference on Supercomputing, (New York, NY, USA), pp. 184–195, ACM, 2004.

[66] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and Performance Using Partitioned Global Address Space]Languages," in PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation, (New York, NY, USA), pp. 24–32, ACM, 2007.

[67] B. Liblit, "Local Qualification Inference for Titanium," Aug. 26 1998. CS263/CS265 semester project report.

[68] R. W. Numrich and J. Reid, "Co-array Fortran for Parallel Programming," SIGPLAN Fortran Forum, vol. 17, no. 2, pp. 1–31, 1998.

[69] U. Consortium, "UPC Language Specifications, v1.2," tech. rep., 2005.

[70] D. Bonachea and J. Duell, "Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations," in 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03), 2003.

[71] D. Bonachea, "GASNet Specification, v1.1," tech. rep., Berkeley, CA, USA, 2002.

[72] W. Gropp and E. Lusk, "Dynamic process management in an MPI setting," pp. 530–533, 1995.

[73] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," Parallel Computing, vol. 22, pp. 789–828, Sept. 1996.

[74] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," in Proceedings of Supercomputing Symposium, pp. 379–386, 1994.

[75] L. Dalcín, R. Paz, and M. Storti, "MPI for Python," J. Parallel Distrib. Comput., vol. 65, no. 9, pp. 1108–1115, 2005.

[76] J. Willcock, A. Lumsdaine, and A. Robison, "Using MPI with C# and the common language infrastructure," in JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, (New York, NY, USA), pp. 238–238, ACM, 2002.

[77] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim, "mpiJava 1.2: API Specification."

[78] E. Ong, "MPI Ruby: Scripting in a Parallel Environment," Computing in Science and Engg., vol. 4, no. 4, pp. 78–82, 2002.

[79] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep. UCB/EECS-2006-18, Electrical Engineering and Computer Sciences, University of California at Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html, December 2006.

[80] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," ACM Computing Surveys, vol. 30, no. 2, pp. 123–169, 1998.

[81] P. B. Hansen, "An Evaluation of the Message-Passing Interface," SIGPLAN Not., vol. 33, no. 3, pp. 65–72, 1998.

[82] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick, "Parallel programming in Split-C," in Supercomputing, pp. 262–273, 1993.

[83] W. W. Carlson and J. M. Draper, "Distributed data access in AC," SIGPLAN Not., vol. 30, no. 8, pp. 39–47, 1995.

[84] K. Datta, "The NAS Parallel Benchmarks in Titanium," Tech. Rep. UCB/EECS-2006-5, EECS Department, University of California, Berkeley, Jan 2006.

[85] G. Balls, "A Finite Difference Domain Decomposition Method Using Local Corrections for the Solution of Poisson's Equation," 1999.

[86] P. McCorquodale and P. Colella, "Implementation of a Multilevel Algorithm for Gas Dynamics in a High-Performance Java Dialect."

[87] G. Pike, L. Semenzato, P. Colella, and P. N. Hilfinger, "Parallel 3D Adaptive Mesh Refinement in Titanium," in PPSC, 1999.

[88] E. Givelberg and K. Yelick, "Distributed Immersed Boundary Simulation in Titanium," SIAM J. Sci. Comput., vol. 28, no. 4, pp. 1361–1378, 2006.

[89] A. Funk, V. Basili, L. Hochstein, and J. Kepner, “Application of a Development Time Productivity Metric to Parallel Software Development,” in SE-HPCS ’05: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, (New York, NY, USA), pp. 8–12, ACM, 2005.

[90] J. Shalf, “The new landscape of parallel computer architecture,” Journal of Physics Conference Series, vol. 78, pp. 2066–+, July 2007.

[91] H. Chhetri and C. Okoye, “Hello, World! Page.” Hello, World! Project.

[92] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, “Productivity Analysis of the UPC Language.”

[93] K. Li and V. Y. Pan, “Parallel Matrix Multiplication on a Linear Array with a Reconfigurable Pipelined Bus System,” IEEE Transactions on Computers, vol. 50, no. 5, pp. 519–525, 2001.

[94] R. P. Brent, “Algorithms for Matrix Multiplication,” Tech. Rep. TR-CS-70-157, Stanford University, Mar 1970.

[95] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans, “Group-theoretic algorithms for matrix multiplication,” Nov 2005.

[96] N. Eiron, M. Rodeh, and I. Steinwarts, “Matrix Multiplication: A Case Study of Algorithm Engineering,” in Proceedings WAE’98, (Saarbrücken, Germany), Aug 1998.

[97] V. Strassen, “Gaussian elimination is not optimal,”

[98] D. Coppersmith and S. Winograd, “Matrix Multiplication via Arithmetic Progressions,” in Proceedings of the 19-th annual ACM conference on Theory of computing, pp. 1–6, 1987.

[99] J. Su and K. Yelick, “Automatic Communication Performance Debugging in PGAS Languages,” tech. rep., October 2007.