Clemson University
TigerPrints

Publications | School of Computing

12-2008

A Performance and Productivity Study using MPI, Titanium, and Fortress

Amy Apon
Clemson University, aapon@clemson.edu

Chris Bryan
University of Arkansas - Main Campus

Wesley Emeneker
University of Arkansas - Main Campus

Follow this and additional works at: https://tigerprints.clemson.edu/computing_pubs

Part of the Computer Sciences Commons

Recommended Citation

Apon, Amy; Bryan, Chris; and Emeneker, Wesley, "A Performance and Productivity Study using MPI, Titanium, and Fortress" (2008). *Publications* . 10.
https://tigerprints.clemson.edu/computing_pubs/10

This Conference Proceeding is brought to you for free and open access by the School of Computing at TigerPrints. It has been accepted for inclusion in Publications by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

A Performance and Productivity Study using MPI, Titanium, and Fortress

Chris Bryan, Wesley Emeneker, and Amy Apon
Department of Computer Science and Computer Engineering (CSCE)
University of Arkansas, Fayetteville
Email: {cjb04, ewe, aapon}@uark.edu

I. Introduction

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 papers presents results from a productivity study of these two newcomers with MPI, the de facto standard for parallel programming [1].

With increased focus on development of programmer-friendly parallel languages, it has become clear that runtime, while important, is no longer the only metric that counts. Programmer productivity should also be considered. This papers presents results of a study of programmer productivity and language usability for the standard MPI and two new developing programming languages, Titanium [2] and Fortress [3]. These new languages have an emphasis not only on performance, but on ease of usability. These results are therefore holistic, as they affect both performance and productivity. These are obtained by coding two algorithms- a computation-intensive matrix multiply and a communication-intensive matrix transformation, in each language to form a program chrestomathy. The resulting programs can be compared against each other.

II. Overview

Distributed-memory programming requires message passing, either explicitly or implicitly through use of a global address space (GAS). GAS programming is generally considered easier to code than explicit message passing models, but may incur performance penalties on distributed hardware. Explicit message passing can be tailored using efficient algorithms for a particular application, and communication can be sent in bulk instead of many small, individual messages that a GAS might inadvertently use. Explicit message passing requires fine-grained program control and careful algorithm setup, which may lead to tedious programming and debugging. However, the explicit message passing paradigm (notably MPI) is still the most widely used way to achieve parallel programming on distributed systems due to its performance.

Currently, many languages are being researched and developed to try to find a balance between the high performance of MPI, and the usability of GAS programming. In this paper, MPI is compared against Titanium and Fortress- two of these newcomers.

a) MPI Overview: The MPI standard was introduced in 1994, and quickly became the *de facto* standard for HPC message passing [4]. MPI has been a very successful approach for achieving parallelism in programs, as it is widespread, portable, and achieves high performance. It has been described as a “complete” model, whereby any parallel algorithm may be implemented [5].

One common issue raised about MPI is its complexity- the high control needed to program using MPI can make for tedious programs that are difficult to debug, or port between hardwares seamlessly.

b) Titanium Overview: Titanium is a language designed for "high-performance parallel scientific computing" designed by the University of California-Berkley [2]. It is based on Java syntax, but at compile time translates code into C and uses an available backend for messaging. Titanium is specifically referred to as a PGAS, or Partitioned GAS, langauge, where global data may be specifically allocated or divided among distributed memory regions. Global data may be implicitly referenced by global pointers, or can be copied by explicit commands (similar to MPI). This allows for two types of programming styles to be used. Programming can be done entirely relying on a GAS, defaulting all variables to global status and able to be accessed by concurrent processes at any time. This may incur performance penalties though. With this in mind, programs (or parts of programs) may be refined using specified data distribution and allocation between processes, cutting down on global lookup and modification costs.

c) Fortress Overview: Fortress is called a "novel" language for HPC. It is a built-from-the-ground-up effort by Sun, looking to become a standard in next-generation, multicore, systems [6]. Fortress was a part of the DARPA HPCS initiative through Phase II, and is now an open-source project. The current compiler runs on top of a Java Virtual Machine, and only a small core of the language specification was working at the time of this testing. The program structure of Fortress is meant to reflect scienctific and mathematical notation, and its syntax, semantics, and parallelism reflect that. Parallelism in Fortress is meant to explicit, with compilers automatically distributing actions like **for** loops or parallel blocks of code. Like Titanium, Fortress will eventually allow PGAS programming, by letting the programmer specify data locality.

d) Kernels: Two simple kernels were implemented and tested- a matrix multiply and matrix transform. The matrix multiply used a straightforward decomposition, if needed, to form submatrices which were multiplied and summed. Such an approach would test heavy computation. The matrix transform is written to illustrate use of heavy interprocess communication, either by a global matrix being accessed by all processes (Titanium and Fortress), or by partitioning the matrix and using ping-pong communication to transform it (MPI).

e) Benchmarks: To test performance and productivity, the kernels were evaluated over a small set of benchmarks. For performance, runtimes and scalability were checked. For productivity, a small set of programmability benchmarks were used to assess the general usability of a program. These benchmarks were defined as:

- Lines of code (LoC), number of characters used (NoC), and characters per line (CpL)
- A sequential-to-parallel conversion effort that describes the effort required to convert code from a sequential base to parallel. This was done for both lines of code, and number of characters. To measure this, two efficiency equations were used:

$$(LoC_{parallel} - LoC_{sequential})/LoC_{sequential} \quad (1)$$

$$(NoC_{parallel} - NoC_{sequential})/NoC_{sequential} \quad (2)$$

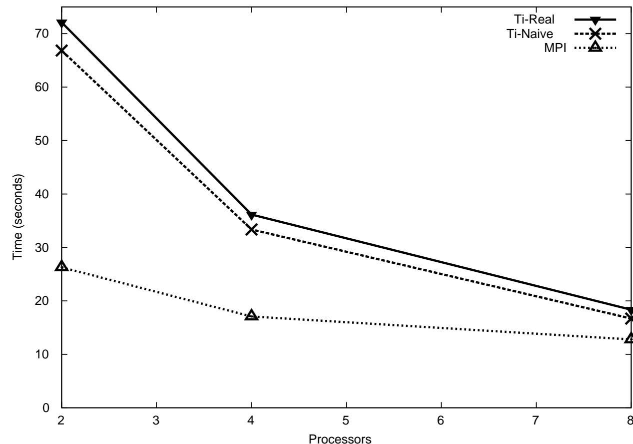
- A parallel conceptual complexity score was generated, based upon language constructs needed to parallelize a program. This evaluates the complexity of parallel calls used, and the number that are required to parallelize each kernel. Different parallel constructs add to a sum score for each kernel.

Regarding the parallel conceptual complexity score, the constructs used were broken up into different categories. Work distributors (WD) allocate work or tasks among processes. Data distributors (DD) describe data itself that must be parallelized or localized. Communicators (Comm) are explicit communication calls. Synchronization

and consistency (SC) calls ensure consistent data between processes. Any other required calls for use of parallel programming are included under miscellaneous (Misc) calls. Each of these calls is further broken up into parameters used, the specific function call itself, and the references in it to processes specific numbers, ranks, or sizes.

f) Testing Setup: The multiply and transform kernels were written in MPI, Titanium, and Fortress. The multiply code was actually written twice in Titanium, once using only GAS programming, and once using explicit data partitioning. For testing sequential-to-parallel conversion effort, MPI was represented in C code, and Titanium in Java. As Fortress is not based on any prior syntax, this test was not applicable.

Fig. 1. SM Multiply 1024x1024 Matrix



III. Results

Both shared memory (SM) and distributed memory (DM) runs were performed for each kernel, with 2, 4, 8, or 16 processors (16 with distributed memory only). At Fortress' current development, true performance testing was impossible, and was thus omitted.

1) Matrix Multiply: For the Matrix Multiply runs, MPI used a matrix partitioning scheme that divided and distributed the matrices among processes. Two Titanium kernels were coded, a "Ti-Nave kernel" using GAS programming, and a "Ti-Real" kernel using explicit data copying.

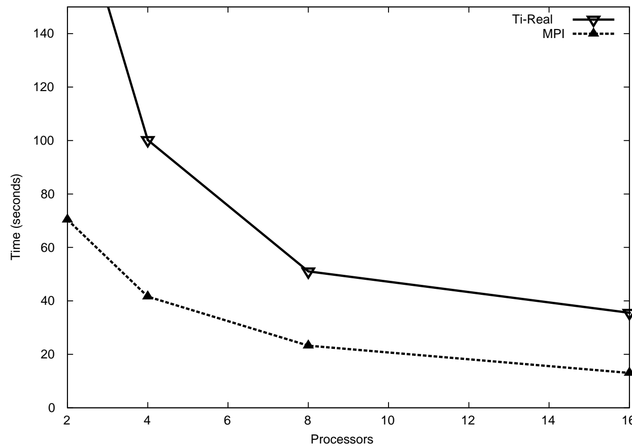
In shared memory, Ti-Real and Ti-Naive had near identical runtimes and scaling on all matrix sizes and number of cores, around a factor of 1.98-1.99, as cores doubled. The MPI code had worse scaling, with limits between a factor of 1.3 to 1.7 as cores doubled. Overall, the MPI code was usually faster than the Titanium codes, with the single exception being at the 2048x2048 matrix size running 8 cores. Compared to Ti-Real, the Ti-Naive was actually slightly faster across all runs (approximately by a factor of 1.09). This is because Ti-Real does local copying of matrices across local memory spaces. Because the copying is local, it is very fast, but the extra work does slow down Ti-Real slightly. In shared memory, a global Titanium call is optimized by the compiler into a local memory operation, simply a memory put or load.

This was not the case for distributed memory however. Running Ti-Naive on distributed memory quickly showed the problems of GAS programming in this hardware environment. With a GAS, every matrix reference became global call, with lookup overhead and communication costs. For matrices of size 128x128, 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 1.14 as cores double (this from 2 to 4 cores). The Ti-Real and MPI code both complete computation in under .3 seconds for the same problem size.

A subset of multiply graphs are shown here, in Figures 1 and 2. These are runtimes of matrices sized 1024x1024. In shared memory, the two Titanium kernels show very similar runtime, with Ti-Naive being slightly faster. In distributed memory, the Ti-Naive runtimes are omitted.

2) Matrix Transform: As in the matrix multiply, in shared memory the Titanium compiler translated global calls into local ones. This meant that the Titanium transform code, which is based on GAS programming, is extremely fast in shared memory, while the explicitly message-passing MPI code is not optimized for shared memory,

Fig. 2. DM Multiply - 1024x1024 Matrix



and is much slower. The Titanium code for shared memory runs averaged approximately 1000 times faster than the MPI code.

In distributed memory, the Titanium runtime was very different because of now-truly global calls. The Titanium compiler cannot optimize these calls into local ones, and so they are much, much slower. Titanium runtimes were much more similar to MPI runtimes. Additionally, the MPI code scaled moderately better than the Titanium code. This is because the Titanium code used a global array in a single nodes memory region, as opposed to the MPI codes partitioned arrays that ping pong messages back and forth. With all processes trying to access only one memory region (even with only two nodes), communication was slowed down and resulted in degraded scaling.

TABLE I
MULTIPLY CONVERSION EFFICIENCIES

	MPI	Ti naive	Ti-Real
LoC % Effort	64.5%	48.28%	110.34%
NoC % Effort	63.22%	55.11%	111.79%

A. Productivity Results

Generally, Titanium and Fortress had much better scores in complexity than MPI over all productivity benchmarks.

TABLE II
TRANSFORM CONVERSION EFFICIENCIES

	MPI	Ti
LoC % Effort	116.98%	43.48%
NoC % Effort	79.61%	66.89%

TABLE III
MPI TRANSFORM PARALLEL COMPLEXITY

	Params	Calls	Rank/Size
WD	23	9	2
DD	7	5	
Comm	42	6	6
SC			
Misc	6	5	2
Sub Totals	78	25	10
Total	113		
Notes : 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			

The lines of code and number of characters used in kernels for these languages were much lower in Fortress and Titanium than in MPI, with MPI code sometimes requiring twice the lines of code than Fortress and Titanium. Tables I and II show the conversion efficiencies of the multiply and transform kernels. The sequential-to-parallel conversion efficiency of Titanium was generally lower than MPI as well, excepting for the Ti-Real kernel. This is because the explicit data distribution in Ti-Real required a large amount of extra work and programming. It is notable to see the difference between Ti-Real and Ti-Naive in Table I. By implementing explicit data management, the Titanium code became even more complex than the MPI code. This kernel did not apply to Fortress, as it did not have a base syntax in which it could be represented.

The parallel complexity of Titanium and Fortress were much lower than MPI as well. In parallel conceptual complexity metrics, MPI is shown to be 2-3 times as complex as Titanium and Fortress for these kernels. This is seen in Tables III, IV and V, which show the conceptual complexity scores for the matrix transform kernels. Note how the Fortress code, in Figure V had only

TABLE IV
TITANIUM TRANSFORM PARALLEL COMPLEXITY

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

TABLE V
FORTRESS TRANSFORM PARALLEL COMPLEXITY

	Params	Calls	Rank/Size
WD	12	4	
DD			
Comm			
SC		1	
Misc			
Sub Totals	12	5	
Total	17		
Notes : 4 for, 1 atomic..do			

two types of calls that are scored here, while the Titanium code had over triple of that number. The MPI code had even more than that, demonstrating that MPI has a high parallel complexity. The matrix multiply scores demonstrated the same pattern, with Fortress being the least complex, followed by Titanium and then MPI (even Ti-Real was less than MPI in this benchmark, although its table is omitted here).

IV. CONCLUSIONS AND FUTURE WORK

Many people have said that MPI is a complex way to perform parallel computation; for these kernels that observation was quantified. Titanium seems to be more usable and programmable than MPI, as it received generally better productivity scores, excepting when it used explicitly distributed data (Ti-Real), when it compared to MPI's complexity. This shows that GAS programming can be a double-edged sword, good for productivity but detrimental to performance. Fortress itself was not evaluated to a final score, although it shows good productivity promise.

There are many avenues for future productivity testing, both for these and other new parallel languages. This study only considered a small set of programmability benchmarks; there are others that could be implemented. Code development time is a very relevant measure of a language's feasibility, as is the learning curve for a new language. Debugging and error checking are not considered here at all, but these are difficult issues for parallel programming. There has been work done, especially in Titanium, on unintended consequences of GAS programming [7], where unforeseen global references can hinder performance. This is a difficult thing to debug for, although developing tools to do this could be very beneficial for future language development.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

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