Performance Comparison of MPI vs. Titanium

Roger Bharath
Stephen Lau

CSE 260
Prof. Scott Baden
Monday, 11 June 2001

1. Introduction

MPI[1] and Titanium[2] support parallel programming but using two different approaches. MPI uses libraries to provide communication procedure calls where as Titanium embeds support at the language level. The result is a contrast between explicit and implied communication. It is interesting to compare the relative performance of MPI and Titanium. To enable a fair comparison, hardware, network interfaces and the algorithms implemented should be kept as similar as possible. The following report details the efforts made in comparing Titanium to MPI. All experiments were conducted on the Rocks Meteor[6] cluster at SDSC

Titanium Language Features

This section has some observations for first-time users of Titanium or even more generally those working within a global address space when accustomed to a message-passing paradigm. With a global address space it is possible to access memories on remote processors. There is a cost to this, both in communication and in maintaining consistency should the values change. Titanium allows pointers to remote memories which sometimes made it difficult to isolate where a program is spending most of its time. Determining the physical location of an object is done upon instantiation.

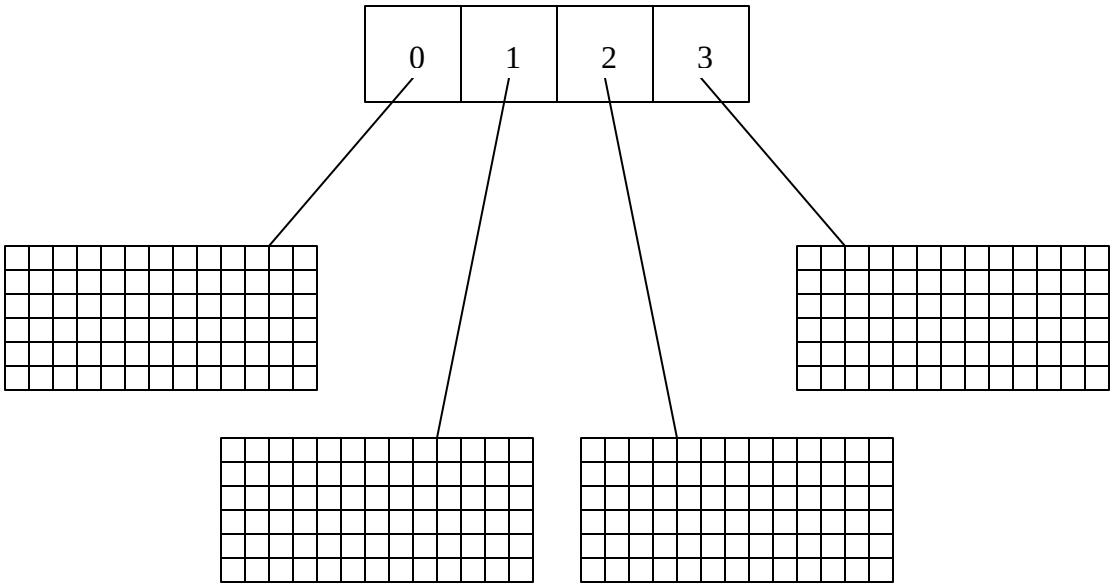
The language reserves two keywords for group communication specifically `broadcast` and `exchange` which correspond roughly to `Bcast` and `Allgather` respectively. Point-to-point communication is possible between processes with a shared data structure. Titanium has a two step compilation process. The first step compiles the Titanium source to a C object code. The C object code is then compiled into a platform specific executable. Thus, it was important to keep the compiler and compiler flags consistent with the MPI version. The compiler used was `gcc` and the optimization level used was `-O3`. Each of the codes used double precision data elements. The compilation option for Titanium assumed one process per processor (i.e. the `udp-cluster-uniprocess` backend, with `TI_PFORP="1/1"`).

2. Two-dimensional Fast Fourier Transform (FFT)

Implementing the Fast Fourier Transform (FFT) was relatively straightforward. The partitioning of the problem was done by rows. The FFT run is independent of any other process, so it was simple enough to find and use a serial FFT calculation, and use that to

run a one-dimensional FFT on each processor's chunk of the 2D grid. We found such an algorithm online[10]. We modified the algorithm to use our basic complex class, as well as to fit the notation of a Titanium two-dimensional array. The only parallel communication in this algorithm occurs in the transpose stage.

The setup of this problem is done using Titanium arrays and domains. A two-dimensional grid is split up into a one-dimensional array of two-dimensional rectangular sub-grids. Conceptually, it looks something like:



The one-dimensional array is declared `single` so Titanium will enforce consistency across it. Each element of that array is then a pointer to some sub-grid residing on a processor. So if processor 2 wants to access element 4,3 of the sub-grid belonging to processor 0, it can access like so: array[0][4,3]. This notation is simple to read, simple to use, and overall just easier to code than explicit message passing as found in MPI. One nice part about this notation is that when processor 2 wants to access the element in processor 0, processor 0 doesn't need to know about it. For example, if we were to code the above example in MPI, processor 0 would need to know that processor 2 wants element [4,3] so it can send it in a message. In Titanium, this is not necessary as the processor can simply access it as remote data. MPI is more explicit about the splitting up of the grid, whereas Titanium treats it more as a distributed grid.

The setup of the grid is done as follows. First, each processor defines a two-dimensional `RectDomain` from [0,0] to [nx,ny], thus creating the concept of a domain of size nx by ny. Then the one-dimensional array is created of size (num_nodes-1). Lastly, each processor creates a two-dimensional complex array fitted to the RectDomain defined earlier; the reference to this array is then broadcast to all other nodes, with the corresponding entry in the one-dimensional array (corresponding to the process id) set to that reference.

The 2D FFT is done by simply doing a 1D FFT over each processor's sub-grid, then doing a transpose, and running the 1D FFT once more. All of the parallelization occurs in the transpose step where the processors need to swap cells back and forth. Initially, we had thought about doing the blocked hierarchical transpose as shown in lecture, and as implemented in our MPI version. However, we thought then there that might be more overhead than is necessary to copy the elements into blocks. Also, in MPI it was advantageous to use the blocked hierarchical transpose due to the message-passing environment. Since Titanium treats the grid as more of a distributed grid, we decided to implement the algorithm using remote accesses. This makes for a much simpler algorithm, as each processor can access an element on another processor's sub-grid just as easily as it access its own.

The transpose step is done by iterating through the element above the diagonal (this algorithm assumes a square grid to begin with), and swapping elements on either side of the diagonal. Initially, this seemed better than the hierarchical transpose, as each element is touched and swapped just once; whereas in the hierarchical transpose method, elements can be swapped multiple times.

We also made use of the `immutable` flag by flagging the complex class (composed of two doubles: real and imaginary) as immutable to force Titanium to swap the values rather than references. This lets the transpose operation take all the burden of communication values (rather than references as mentioned) so that the second FFT doesn't have to be constantly accessing remote nodes of the cell. Essentially, flagging a class as immutable makes the Titanium compiler treat the class similar to a C struct.

Below are the tabulations of the raw FFT performance times.

Table 2.1 – FFT Running Times

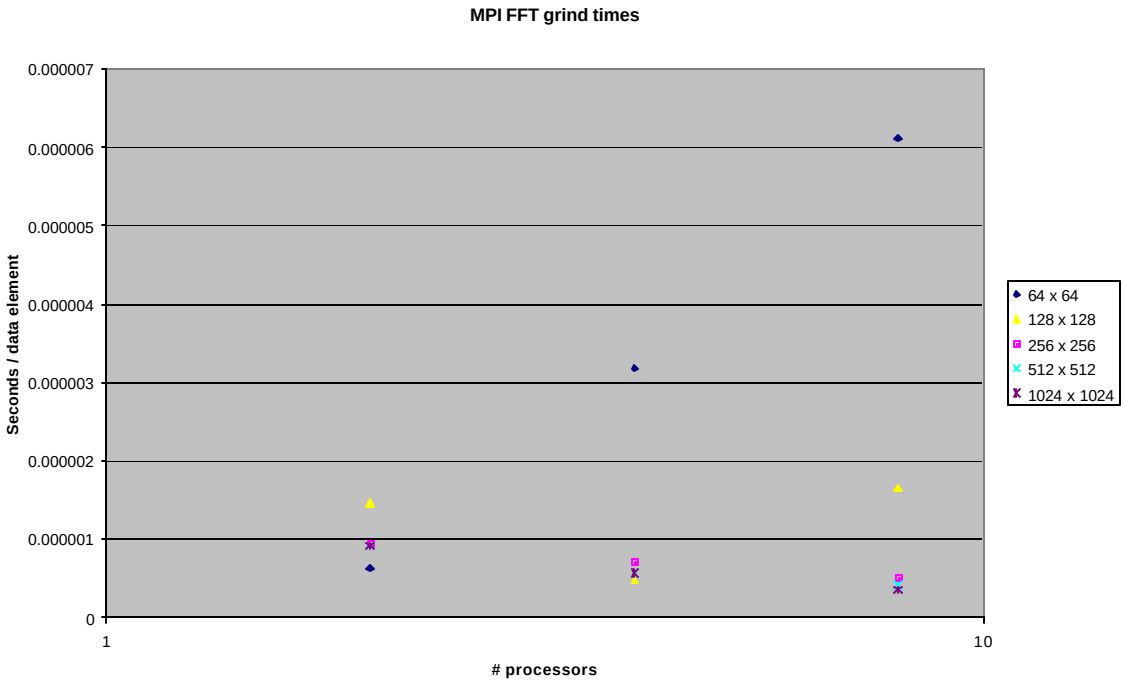
Data size	Processors	MPI	Titanium
64x64	2	0.0026	0.34
64x64	4	0.013	0.24
64x64	8	0.025	0.14
128x128	2	0.024	1.39
128x128	4	0.008	0.99
128x128	8	0.027	0.57
256x256	2	0.061	5.68
256x256	4	0.047	3.97
256x256	8	0.033	2.31
512x512	2	0.24	23.3
512x512	4	0.15	16.2
512x512	8	0.11	9.39
1024x1024	2	0.96	95.4
1024x1024	4	0.60	65.8
1024x1024	8	0.38	37.9

Table 2.2 – FFT Running Times w/o Communication

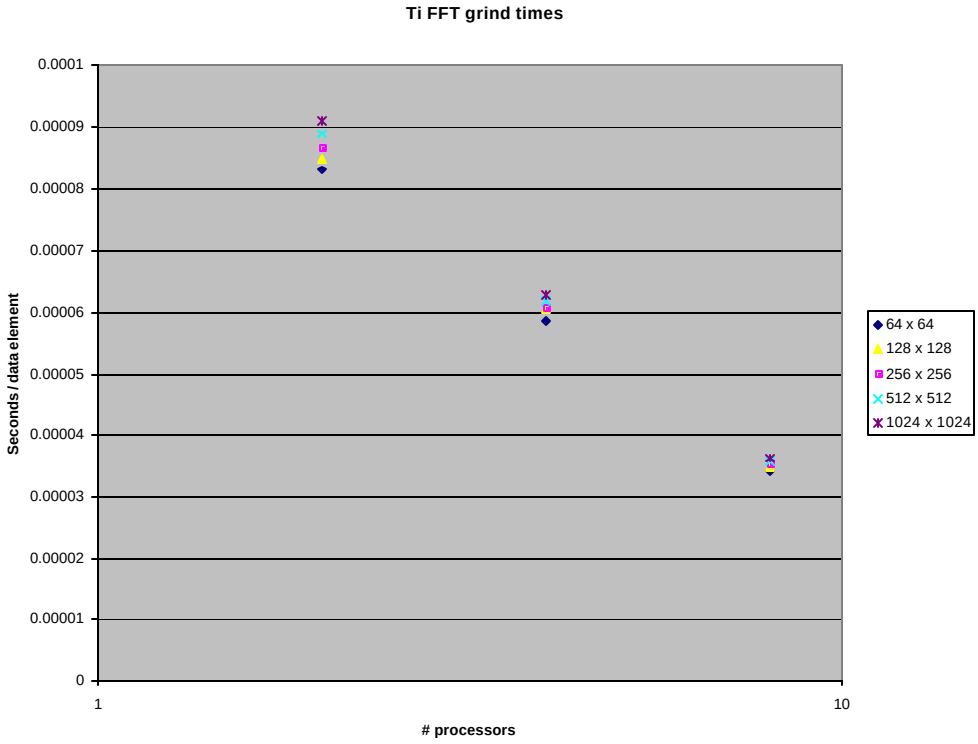
Data size	Processors	MPI	Titanium
64x64	2	0.0026	0.056
64x64	4	0.013	0.029
64x64	8	0.025	0.015
128x128	2	0.011	0.254
128x128	4	0.0056	0.128
128x128	8	0.0032	0.064
256x256	2	0.061	1.15
256x256	4	0.047	0.58
256x256	8	0.033	0.29
512x512	2	0.208	5.14
512x512	4	0.114	2.57
512x512	8	0.065	1.28
1024x1024	2	0.96	22.7
1024x1024	4	0.60	11.4
1024x1024	8	0.38	5.7

The FFT graph has poor performance across the entire spectrum of processor-data size combinations. The data collected with the FFT running with no communication identifies the communication as the primary factor in the running time for FFT. The code currently is using a hierarchical transpose operation which may benefit from some kind of bundling operation. The reason for this is that the transpose operation bay be sending a lot of small "element-sized" messages. This would result in high overhead which could be alleviated by grouping elements together in fewer messages.

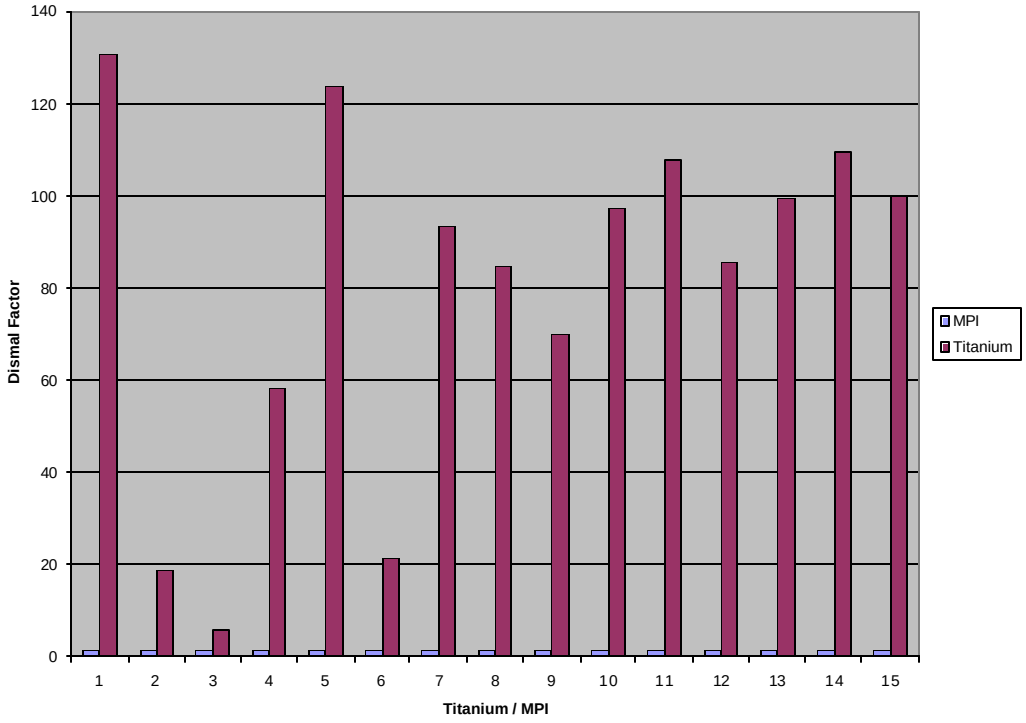
Graph 1



Graph 2



Graph 3



The plots of the grind times show some peculiarities. In particular, note that the 64 x 64 data array is not representative of the overall trend. This array size is at an extreme end of the range and is not as useful a data size to partition. So we choose to ignore this as an outlier. The remaining points still suggest improved performance per data element as processor size scales.

3. Three-dimensional Red-Black Successive Over-relaxation (SOR)

The MPI code used as the reference implementation was the one provided for the class on the class webpage [7], and was implemented mainly in C++ with the relaxation code written in Fortran-77.

The algorithm, as implemented in Titanium, was completed in several phases. We first studied the algorithm as presented in Jim Demmel's [8] online notes, and built a rough framework around that. We then wrote a serial version of the algorithm in pure Java. Once this was completed, work was done in order to parallelise it and port it to Titanium. The relaxation function was ported mainly from the Fortran-77 relaxation code used in the MPI algorithm given, all other portions of the Titanium implementation were written from scratch.

In designing the algorithm, the visualisation of the problem was one of the harder design issues to overcome. This quarter's parallel programming assignments have all been exclusively in MPI using C/C++. Trying to shift gears from a message passing mentality to the programming model used by Titanium was initially hard. Instead of having exclusive control through the sending and receiving of messages, Titanium gets its parallelism through the use of domains, and distributing memory. It seems that its target application area is the domain of problems using a grid-like structure, so we hoped that the performance of red-black 3D would be good.

The way we structured the data was similar to the way we implemented the FFT code. We again had a one-dimensional array of references to localised three-dimensional sub-cubes (instead of the two-dimensional sub-grids found in FFT). Also similar to the FFT, the computation portion of the code was all localised within the relax() function. The only communication occurred in the ghost-region filling, where copying over remote data is done by simply indexing first the processor within the one-dimensional array, and then referencing the element within that sub-cube.

It turns out that our Titanium performance results were considerably slower than the MPI code. The default run of the MPI code was to do a 256x256x256 cube, and partition it along the z-axis. We decided to run a base run using the default problem size and default partitioning with four processors. This resulted in the following measurements:

Table 3.1 - Results for MPI, 256x256x256, 4 processors

Time to Relax (s)/iteration	Communication Time/iteration	Time per Iteration(s)	MFLOPS
1.47361	1.1864	2.43955	26.344

Running the equivalent Titanium benchmark yields the following measurements:

Table 3.2 – Results for Titanium, 256x256x256, 4 processors

Time to Relax (s)/iteration	Time per Iteration (s)	MFLOPS
63.859124	248.31423	0.00128869

As one can see from these results, performance was anaemic at best, giving a 43-fold increase in relaxation time, and a 101-fold increase in iteration time. Communication time was not measured due to the lack of explicit communications calls to wrap timers around.

These measurements, needless to say, were not what we expected. Putting in some diagnostic output, we saw that the iteration times varied considerably. For instance, the range of relaxation times varied from as little as 17.3819 seconds per relaxation-call to as much as 34.58235 seconds. Adding to this, the difference between calls varied by as much as approximately 38 seconds (i.e.: processor p_0 would initiation a relaxation phase, and 38 seconds, another processor p_1 would being initiating its own call when p_0 had just finished completing!). Obviously, parallelism is not being achieved… upon further investigation, it seemed that some of the nodes were being more utilised than others which led to the high variance in runs. Some processors were still initialising their arrays while others had gone on to already start the relaxation phases. Running the benchmarks at off-peak times, on less-utilised nodes resulted in more even distributions of ~27 seconds per relaxation phase. However, there was still a high variance in the initial call to the relaxation function, on the order of 15-30 seconds. This resulted in the times below:

Table 3.3 – Results for Titanium, 256x256x256, 4 processors

Time to Relax (s)/iteration	Time per Iteration (s)	MFLOPS
54.389735	214.748567	0.00149012

This was a slowdown of 36 and 88-fold respectively for the relaxation and iteration times.

Our next step was to try and isolate why Titanium was running the program so much slower than MPI. At first our initial thought was that maybe the triply-nested for loops were being run in the wrong order. Since they had been ported from Fortran-77, we thought maybe the column-major order was causing a slowdown. We ran some tests on different sized arrays, and noted that we seemed to get the best performance when iterating in the y-axis, then the x-axis, and iterating over the z-axis as the inner loop. However, this didn't fix the slowdown we were observing. Our next thought was that maybe we were hitting bad strides in the iterations, and that blocking for cache might help. We ran the algorithm on a small (10x10x10) problem size, and noted that it still ran much slower than MPI. At this point we went into the relaxation phase to try and determine what was causing the slowdown. Apparently, it was the memory references to the arrays which were causing the slowdown. Taking those out caused the algorithm to run comparable to MPI, so we put back in the array references one by one, and we noted that as we put them back in, the running times scaled linearly with the number of

references we put in. (i.e.: the running times doubled if we put in two references, and tripled if we put in three).

At this point, the only thing we were left to conclude was that possibly Titanium was distributing the grids non-locally, i.e.: instead of processor 0 owning U[0][x,y,z] we thought perhaps Titanium was causing that sub-cube to be stored on another processor's memory. We tested this by calling the `isLocal()` method that can be called on any reference object in Titanium. The result showed that the sub-cubes were being properly distributed so that locality was being optimised.

Our last step was to try different optimisations (as suggested by Greg Balls). The different flags we tried were –nobcheck (disable array bounds checking), --optimize (optimise both the given Titanium code, as well as the generated C code), and –cc-flags –O9 (passes optimisation level 9 to the underlying cc C compiler). This resulted in huge increases in performance. Running the algorithm after the optimisations, we observed the following measurements:

Table 3.4 – Results for Titanium (optimised), 256x256x256, 4 processors

Time to Relax (s)/iteration	Time per Iteration (s)	MFLOPS
12.140048	73.944381	0.004327758

We noticed that, with these optimisations, each red-black relaxation phase went from being ~30 seconds to being between 6-7 seconds.

Using the optimisations, we got the following results for various pool sizes:

Table 3.5 – Results for Titanium (optimised), 256x256x256

Processors	Time to Relax/Iter.	Time per Iteration	MFLOPS	Total Time
1	68.342651	68.39977	0.0046784	202.830517
2	32.615337	54.550878	0.0058661	180.382921
4	16.387337	54.032965	0.0059223	181.005489
8	8.032683	49.567715	0.0064558	129.954172
16	3.933256	43.408371	0.0073719	104.900989

note: Total Time is the total time for the iterative part of the algorithm, and does not include initialisation and setup code. Time to Relax/Iter is the total relaxation time per iteration, or approximately twice (one for red, and one for black) the relaxation time as seen in ***Table 6***

In these runs, the variance increased with the number of processes. Table 6 below lists the minimum and maximum relaxation (per phase) times for each run.

Table 3.6 – Minimum & Maximum Relaxation Times

Processors	Minimum Relax Time	Maximum Relax Time
1	32.74487	33.296722
2	10.543772	16.285903
4	5.253719	8.197816
8	2.283437	4.187135
16	1.163902	1.993602

Looking at these results, it's obvious there was a discrepancy between the node's loads to cause the run times for each relaxation phase to be off by almost 100%. Due to the different loads experienced by each node, and the difference in relax times, each node also started the relaxation phases at different times. However, despite these problems, there was a speedup between processors. Table 7 below lists the speedups as compared to both the single uni-processor system, and the system before it (i.e.: processors/2)

Table 3.7 – Speedups for the Titanium System

Processors	Speedup vs. 1-proc	Speedup vs. P/2
1	1	1
2	1.124444132	1.124444132
4	1.120576609	0.996560502
8	1.560784959	1.392840924
16	1.933542466	1.238826948

Obviously, these are less than desirable results. The problem is not scaling at all; ideally, there should be a speedup of 2 between each system and the system before it, however, the speedups don't even exceed 1.4. The tables below list the corresponding results for the MPI system.

Table 3.8 – Results for MPI (256x256x256)

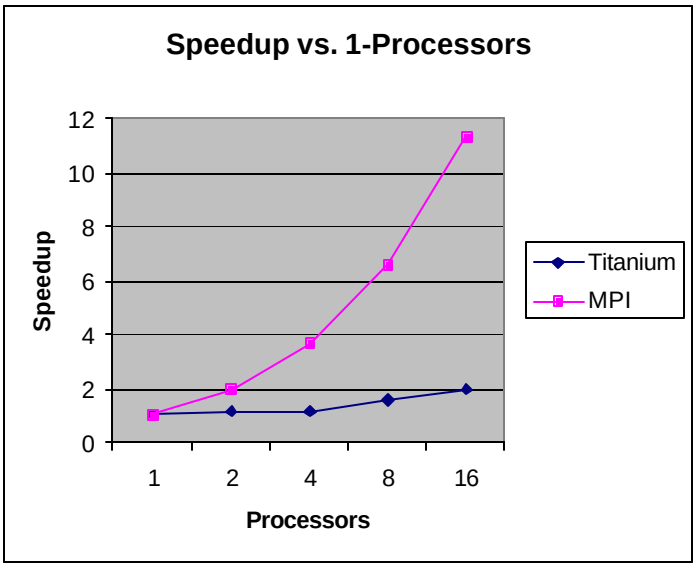
Processors	Time to Relax/Iter.	Time per Iteration	MFLOPS	Total Time
1	5.78392	5.78393	23.2053	11.5583
2	2.92686	2.98281	44.9971	5.96719
4	1.51759	1.58484	84.6887	3.17078
8	0.753588	0.87254	153.824	1.76615
16	0.375061	0.500842	267.984	1.02031

Table 3.9 – Speedups for the MPI System

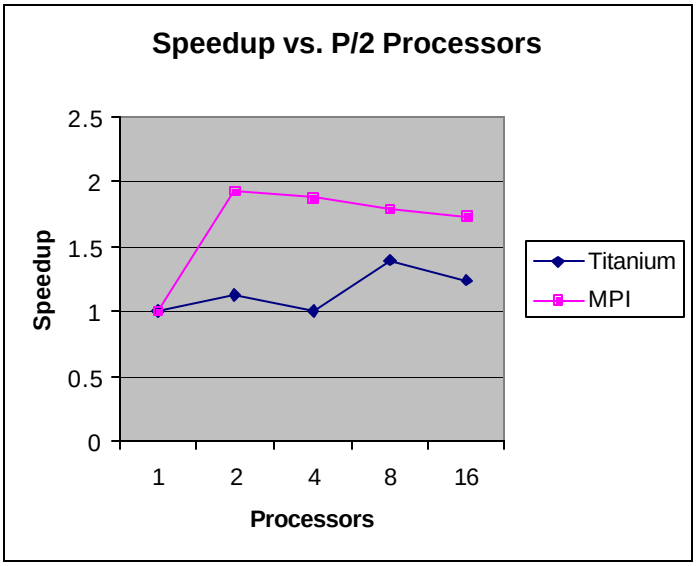
Processors	Speedup vs. 1-proc	Speedup vs. P/2
1	1	1
2	1.936941843	1.936941843
4	3.645254479	1.881931259
8	6.544347875	1.795306174
16	11.32822377	1.730993521

From these results, it is obvious the MPI system far outperforms the Titanium system. While it doesn't achieve linear speedup, one can see that each successive pool-size speedup is close to 2 (1.73 being the lowest). Plotting the speedups of the MPI vs. Titanium, we get a plot like the following:

Graph 3.1



Graph 3.2



In Tables 10 & 11, and Graphs 3 4 4 below, we have plotted the grind times for both MPI and Titanium to observe the amount of work per data element at different problem-sizes.

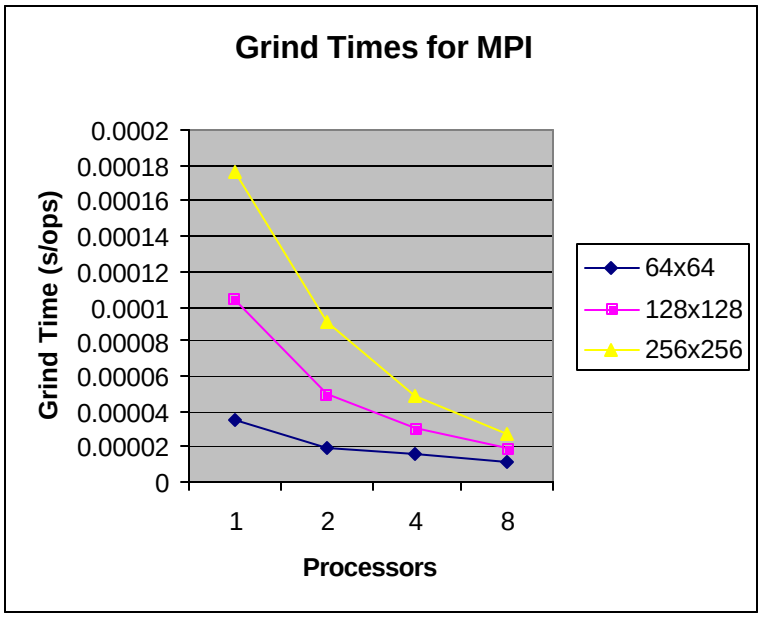
Table 3.10 – Grind time for MPI Red-black 3D

Problem Size	Running Time			
	1	2	4	8
64x64	0.1453	0.0781	0.0651	0.0441
128x128	1.7053	0.8154	0.5014	0.3093
256x256	11.5585	5.9672	3.1712	1.7671

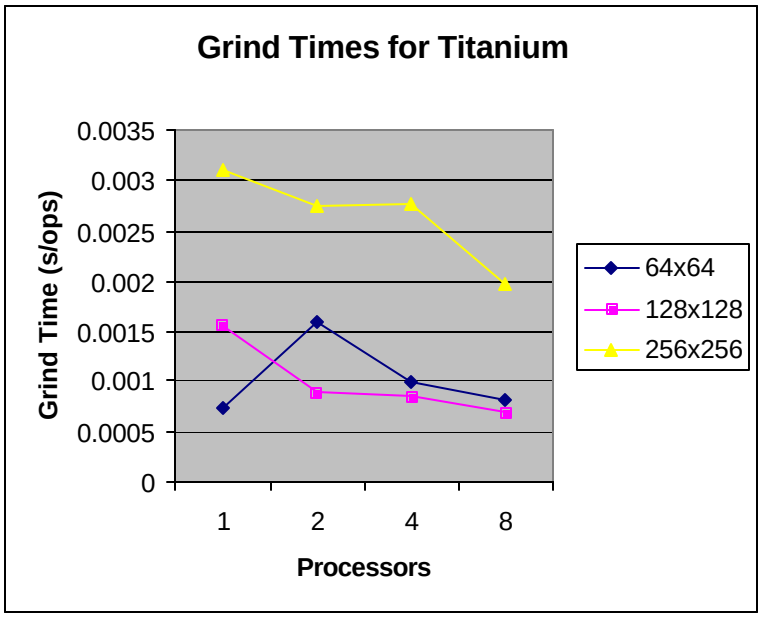
Table 3.11 – Grind time for Titanium Red-black 3D

Problem Size	Running Time			
	1	2	4	8
64x64	3.0271	6.546731	4.0565	3.2993
128x128	25.485	14.454383	13.8242	11.3465
256x256	202.831	180.383	181.006	129.954

Graph 3.3



Graph 3.4



Titanium is not only orders of magnitude slower than MPI, it also doesn't scale nearly as well as the MPI benchmark. However, we believe there are two primary reasons for this. The first being that the MPI program was a finely tuned benchmark, optimised to run as

fast as possible. Our Titanium coded version is undoubtedly not as well optimised as could be. A programmer fluent and more knowledgeable about Titanium could undoubtedly write a better red-black 3D benchmark.

The second reason, and this could be a huge factor is that the MPI execution run-time enjoys access to the Myrinet network connecting the clusters. From the Myricom homepage [9], "Myrinet is a cost-effective, high-performance, packet-communication and switching technology that is widely used to interconnect clusters of workstations…" The Myrinet network is an extremely high-speed network that far surpasses convential Ethernet. Speedwise, Myrinet allows links up to 2+2 (full-duplex) GB/s, whereas the Ethernet also connecting the nodes is a standard 100 MB/s. Secondly, the Myrinet is a lower-latency network than the Ethernet. The Titanium runtime is limited to using the standard Ethernet to do its communications.

The Myrinet certainly boosts the performance of the MPI process in terms of communications (i.e.: filling the ghost cell regions). However, we note that the relaxation/iteration times, which are done completely on local sub-cubes, show that Titanium relaxation times are still about 11 times slower than the MPI relaxation times.

To do a more comparable benchmark, the MPI runtime should be forced to use the Ethernet as well, we looked into how to force this – but were unable to run MPI on the Ethernet, as all the libraries and packages were installed system-wide to use the Myrinet back-end.

4. Conjugate Gradient

Conjugate Gradient is an iterative method that use gradients and the method of steepest descent. Computationally, the method uses BLAS 1 and BLAS 2 building blocks. In addition to the data array (A), right hand side (b) and current guess (x), the algorithm uses 3 additional vectors to compute a gradient with respect to the error function. The basic structure of the algorithm[12] is presented here:

```
Given:      2d square positive definite array A
            vector b
            x is an initial guess
      while (r2 > epsilon)
            r = p = b – Ax
            q = A p                                  // BLAS 2
            r2 = r^T • r   (r transpose dotted with r)   // BLAS 1
            alpha = r2 / (p^T • q)
            r = r - alpha q                          // BLAS 1
            x = x + alpha p                          // BLAS 1
            r2_old = r2
            r2 = r^T dot r                           // BLAS 1
            beta = r2 / r2_old
            p = r + beta p                           // BLAS 1
```

An MPI implementation of this code needs to be aware of global reductions in order to compute dot products of vectors that are distributed over more than one processor. Similarly the matrix multiply needs an `MPI_Allgather` call to assemble pieces of its operand vector. Finally, a barrier synchronizes each iteration, so that processors don't get ahead of each other and read stale values.

The Titanium implementation follows closely along the same lines as the MPI version. It is possible to avoid the global reductions by referencing data elements from other processors directly. The global address space allows this to happen. However this doesn't scale well with an increase in the data size. Especially, if the accesses are made individually as would be the case in a tightly nested loop of the form: "for () sum += a[i]". Hence the reductions have remained.

The timings collected represent the time directly after the data has been dispersed to the processors to the point right before the data is gathered from the processors. The implementation of the Conjugate Gradient MPI code was adapted from a Fortran-90 code[11]. The code was run for a fixed number of iterations (2) to ensure that a constant amount of work was done.

Below are the performance measures we measured for the Conjugate Gradient benchmark.

Table 4.1 – CG Running Times

Data size	Processors	MPI	Titanium
64x64	2	0.001	0.575
64x64	4	0.012	0.330
64x64	8	0.012	0.281
128x128	2	0.103	2.26
128x128	4	0.078	1.24
128x128	8	0.065	1.00
256x256	2	0.250	9.03
256x256	4	0.192	4.79
256x256	8	0.134	3.74
512x512	2	0.384	35.5
512x512	4	0.360	18.9
512x512	8	0.299	14.8
1024x1024	2	0.492	141.0
1024x1024	4	0.405	75.3
1024x1024	8	0.381	58.8

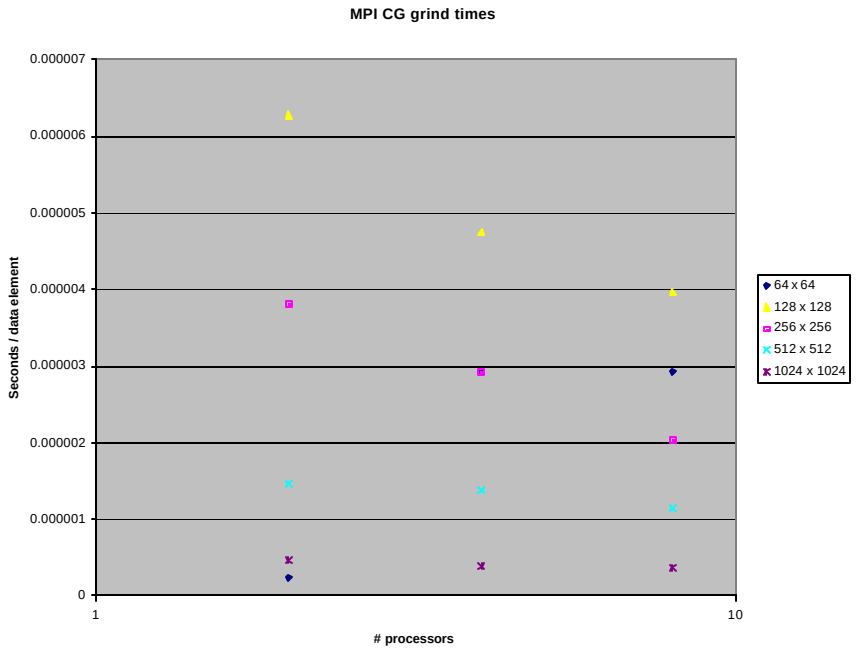
Table 4.2 – CG Running Times w/o (Matrix Multiply) Communication

Data size	Processors	MPI	Titanium
64x64	2	0.001	0.553
64x64	4	0.008	0.315
64x64	8	0.008	0.253
128x128	2	0.090	1.64
128x128	4	0.055	1.19
128x128	8	0.052	2.18
256x256	2	0.223	8.80
256x256	4	0.176	4.67
256x256	8	0.117	3.68
512x512	2	0.339	35.3
512x512	4	0.305	18.8
512x512	8	0.252	14.5
1024x1024	2	0.447	139.8
1024x1024	4	0.358	74.9
1024x1024	8	0.327	58.3

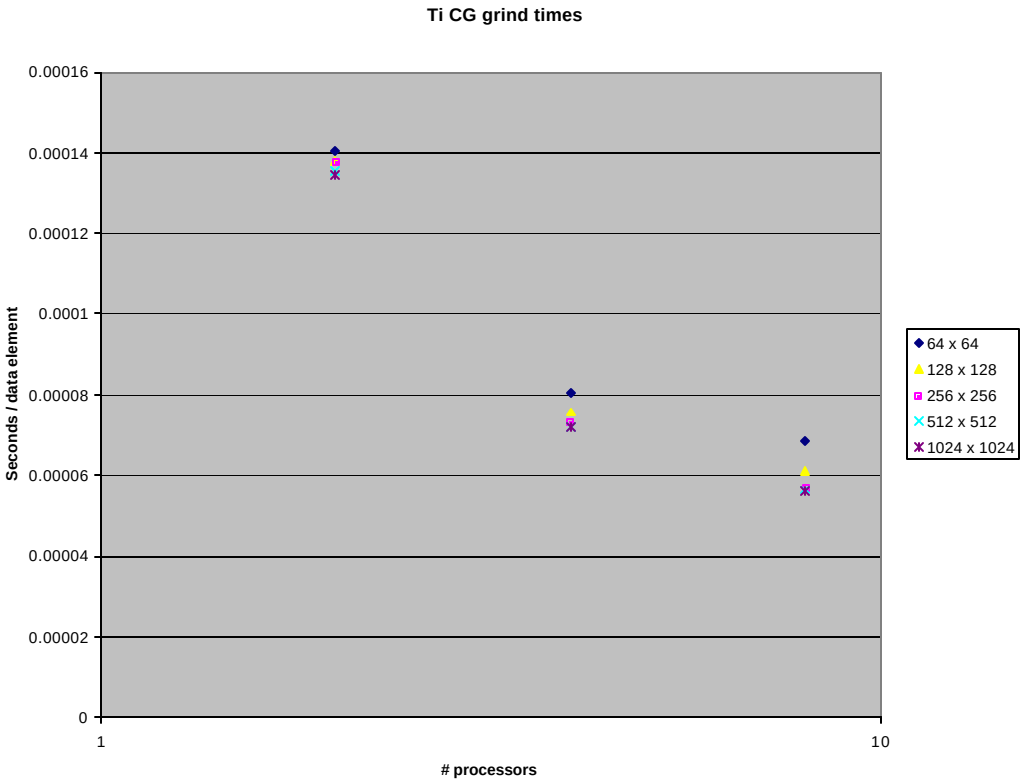
Unfortunately, the analysis presented here is overshadowed by the results. Here are some explanations for the large disparity in performance. To explain the relative performance of the CG code some timing tests were run. Two broadcasts were identified as taking over 90% of the running time.

The variable that the broadcast is performed on is flagged with the "single" compiler directive. From this it would seem that the program is spending an inordinate amount of time maintaining memory consistency when this broadcast occurs. What probably is happening is copies of the entire array are being shipped to every processor. What was desired was to send a pointer to the array to every processor. This alternative would allow each processor to read the elements they need from the array remotely.

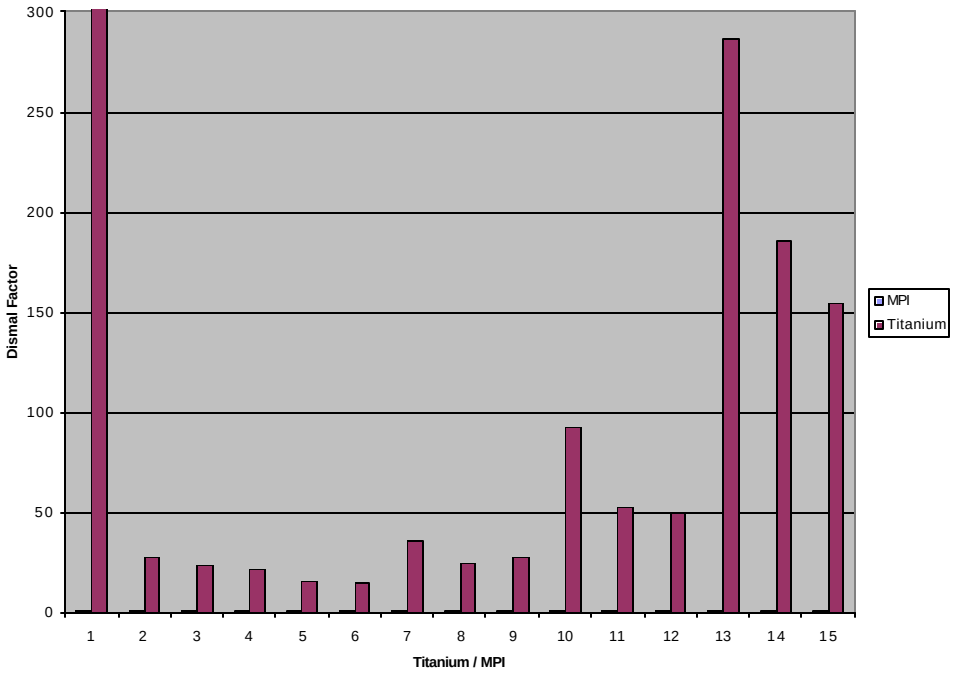
Graph 1



Graph 2



Graph 3



A look at the grind times above in Graphs 1, 2, & 3 for the CG code show that the implementation is clearly not scalable.

References:

[1] MPI (Message Passing Interface) Home Page. http://www.mcs.anl.gov/mpi.

[2] Titanium Home Page. http://www.cs.berkeley.edu/projects/titanium

[3] K. Yelick, L. Semanzato, G. Pike, C. Miyamoto, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken. Titanium: A High Performance Java Dialect. UC Berkeley and Lawrence Berkeley National Laboratory: Special Issue, 1998

[4] P.N. Hilfinger, et al. Titanium Language Reference Manual. Version 0.26. http://www.cs.berkeley.edu/projects/Titanium/doc/lang-ref.ps. April, 2001.

[5] Peter Pacheco. A User's Guide to MPI. University of San Francisco. March, 1998.

[6] NPACI. Meteor – A Rocks Cluster. http://rocks.npaci.edu/meteor.

[7] Scott Baden. CSE 260 Web page – Red Black 3D. http://www-cse.ucsd.edu/~baden/cse260_sp01/Code/rb3d.html

[8] Jim Demmel. CSE 267 Lecture Notes. http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html

[9] Myricom, Inc. Myricom Home Page. http://www.myri.com

[10] Jeffrey D. Taft. The Java FFT Source Code Page. http://www.nauticom.net/www/jdtaft/JavaFFT.htm

[11] Don Cross. Fast Fourier Transform C Library. http://www.intersrv.com/~dcross/fft.html

[12] Lyle Long. Fortran-90 Conjugate Gradient Implementation. http://www.personal.psu.edu/faculty/l/n/lnl/424/mpi/mpi1.html http://www.personal.psu.edu/faculty/l/n/lnl/424/cg.html