The NAS Parallel Benchmarks in Titanium

by Kaushik Datta

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Katherine A. Yelick
Research Advisor

(Date)

* * * * * * *

Professor Paul N. Hilfinger
Second Reader

(Date)

The NAS Parallel Benchmarks in Titanium

Kaushik Datta
kdatta@cs.berkeley.edu
Computer Science Division,
University of California at Berkeley

Abstract

Titanium is an explicitly parallel dialect of Java™ designed for high-performance scientific programming. It offers object-orientation, strong typing, and safe memory management in the context of a language that supports high performance and scalable parallelism. We present an overview of the language features and demonstrate their use in the context of the NAS Parallel Benchmarks, a benchmark suite of common scientific kernels. We argue that parallel languages like Titanium provide greater expressive power than conventional approaches, thereby enabling more concise and expressive code and minimizing time to solution. Moreover, the Titanium implementations of three of the NAS Parallel Benchmarks can match or even exceed the performance of the standard Fortran/MPI implementations at realistic problem sizes and processor scales, while still using far cleaner, shorter and more maintainable code.

Contents

1 Introduction

Titanium is an explicitly parallel dialect of Java designed for high performance and programmability for scientific computing. Titanium leverages Java's object-oriented programming model to allow programmers to build complex irregular data structures, which are increasingly important in scientific applications. For instance, there are Titanium implementations for Adaptive Mesh Refinement (AMR) [28], a contractile torus simulation [18], and an Immersed Boundary simulation (for biological applications) [16].

Titanium also extends Java with a powerful multidimensional array abstraction, lightweight objects, and a template mechanism for parameterized types. For large-scale parallelism, it replaces Java's thread model with a Single Program Multiple Data (SPMD) style of parallelism in which a fixed number of threads are created a program startup. In addition, the language and associated libraries have support for global barrier synchronization and collective communication, as well as implicit communication through a shared address space. The shared address space is partitioned into spaces that are logically associated with each thread, so that application writers can control data layout and the language implementation can efficiently map these layouts onto distributed memory and shared memory hardware. Titanium runs on most single and parallel processor machines, from laptops and desktop machines to shared memory multiprocessors and clusters of networked computers.

In this paper we present three case studies of application benchmarks written in Titanium, using each to highlight some of the language features and performance charac-

teristics of the language. The benchmarks are taken from the NAS Parallel benchmark suite [1] and reflect three important computational methods used in scientific computing: stencil operations on a rectangular 3D mesh (Multigrid, or NAS MG); iterative solvers on a sparse matrix (Conjugate Gradient, or NAS CG); and spectral methods on a regular 3D mesh (Fourier Transform, or NAS FT). The language analysis presented in this paper previously appeared in a joint paper with Bonachea and Yelick [11]. This report gives a more detailed analysis of the performance of each benchmark, illustrating some of the optimization techniques that Titanium programmers can use and how they affect performance. We present performance data on serial, parallel, and communication performance on several different cluster-based machines, and also compare the performance to standard implementations written in Fortran with MPI [19].

Our results show that Titanium programs are significantly shorter than those written in Fortran with MPI, and we present some program fragments to highlight the elegance of the Titanium solutions. These programmability advantages are especially noticeable on applications that extensively use the Titanium array facilities. The performance story is more complicated, but overall quite positive. The global address space model offers some performance advantages relative to two-sided message passing, and the Titanium compilation strategy of translating serial portions of the code to C works very well on some platforms, but is highly dependent on the quality of the backend C compilers. We will give a detailed analysis of the performance of each benchmark and identify some performance tradeoffs that are revealed from these case studies.

2 Experimental Methodology

In order to compare performance between languages, we measured the performance of the Titanium and Fortran with MPI implementations on four platforms: an Itanium 2 cluster with a Myrinet interconnect, an Opteron cluster with InfiniBand, an Alpha cluster with Elan3, and a G5 cluster with InfiniBand. Complete platform details are in appendix A. In addition, appendix B has details concerning input sizes for each benchmark's problem classes. In this paper, class A (the smallest problem size) was used for serial and small parallel runs, while classes C and D (larger problems) were used for larger parallel runs.

During data collection, each data point was run three times on the same set of nodes, with only the minimum times being reported. In addition, for a given number of processors, the Fortran and Titanium codes were both run on the same nodes (to ensure consistency). In all cases, performance variability was low, and the results are reproducible.

3 Multigrid (MG)

The NAS MG benchmark performs multigrid calculations over a regular cubic domain in order to iteratively solve the Poisson equation. The main data structure is a hierarchy of grids, where each grid level represents the same physical domain at varying degrees of discretization. In order to move between these levels, grids can be either coarsened or prolongated. The multigrid computation descends from the finest grid level to the coarsest and back in what is known as a *V-cycle*. At each grid level, nearest-neighbor (stencil) operations are done at each point.

These 3D grids are partitioned across processors by first halving the most significant dimension, then the next most significant dimension, and finally the contiguous dimension in each sub-grid. This process is then repeated until there is a sub-grid for every processor. This procedure limits the processor count to powers of two.

Since stencil operations are performed at every point in a sub-grid, the border points may need values that are located on remote sub-grids. In order to handle this, one-deep layers of points called *ghost cells* are allocated around each processor's sub-grids. These ghost cells are used to locally store the remote values needed by each sub-grid's border points, thereby eliminating the need for expensive fine-grained communication. In addition, the entire domain is periodic, so processors that own blocks at the edge of the global domain communicate with processors at the other edge of the domain in a torus-like fashion. In this case, ghost cells are used to enforce these periodic boundary conditions.

3.1 Titanium Features in the Multigrid Benchmark

This section describes the main Titanium features used in the NAS MG benchmark, including Titanium's multidimensional array abstraction, distributed arrays, and static annotation of data locality.

3.1.1 Titanium Arrays

The NAS benchmarks, like many scientific codes, rely heavily on arrays for the main data structures. The three benchmarks use different types of arrays: CG uses simple 1D arrays to represent the vectors and a set of 1D arrays to represent a sparse matrix; both MG and FT use 3D arrays to represent a discretization of physical space.

Titanium extends Java with a powerful multidimensional array abstraction that provides the same kinds of subarray operations available in Fortran 90. Titanium arrays are indexed by *points* and built on sets of points, called *domains*. Points and domains are first-class entities in Titanium – they can be stored in data structures, specified as literals,

passed as values to methods and manipulated using their own set of operations. For example, the class A version of the MG benchmark requires a 256^3 grid with a one-deep layer of surrounding ghost cells, resulting in a 258^3 grid. Such a grid can be constructed with the following declaration:

```
double [3d] gridA = new double [[-1,-1,-1]:[256,256,256]];
```

The 3D Titanium array `gridA` has a rectangular index set that consists of all points $[i, j, k]$ with integer coordinates such that $-1 \leq i, j, k \leq 256$. Titanium calls such an index set a *rectangular domain* with Titanium type `RectDomain`, since all the points lie within a rectangular box. Titanium arrays can only be built over `RectDomains` (i.e. rectangular sets of points), but they may start at an arbitrary base point, as the example with a $[-1, -1, -1]$ base shows. This allows programmers familiar with C or Fortran arrays to choose 0-based or 1-based arrays, based on personal preference and the problem at hand. In this example the grid was designed to have space for ghost regions, which are all the points that have either -1 or 256 as a coordinate.

The language also includes powerful array operators that can be used to create alternative views of the data in a given array, without an implied copy of the data. For example, the statement:

```
double [3d] gridAIn = gridA.shrink(1);
```

creates a new array variable `gridAIn` which shares all of its elements with `gridA` that are not ghost cells. This domain is computed by shrinking the index set of `gridA` by one element on all sides. `gridAIn` can subsequently be used to reference the interior elements of `gridA`. The same operation can also be accomplished using the `restrict` method, which provides more generality by allowing the index set of the new array view to include only the elements referenced by a given `RectDomain` expression, e.g.: `gridA.restrict(gridA.domain().shrink(1))`, or a using `RectDomain` literal: `gridA.restrict([[0,0,0]:[255,255,255]])`.

Titanium also adds a looping construct, `foreach`, specifically designed for iterating over the points within a domain. More will be said about `foreach` in section 5.1.1, but here we demonstrate the use of `foreach` in a simple example, where the point `p` plays the role of a loop index variable:

```
foreach (p in gridAIn.domain()) {
   gridB[p] = applyStencil(gridA, p);
}
```

The `applyStencil` method may safely refer to elements that are one point away from `p` since the loop is over the interior of a larger array. This one loop concisely expresses iteration over multiple dimensions, corresponding to a multi-level loop nest in other languages.

A common class of loop bounds and indexing errors is avoided by having the compiler and runtime system keep track of the iteration boundaries for the multidimensional traversal.

3.1.2 Stencil Computations Using Point Literals

The stencil operation itself can be written easily using constant offsets. At this point the code becomes dimension-specific, and we show the 2D case with the stencil application code shown in the loop body (rather than a separate method) for illustration. Because points are first-class entities, we can use named constants that are declared once and re-used throughout the stencil operations in MG. Titanium supports both C-style preprocessor definitions and Java's final variable style constants. The following code applies a 5-point 2D stencil to each point `p` in `gridAIn`'s domain, and then writes the resulting value to the same point in `gridB`.

```
final Point<2> NORTH = [0,1], SOUTH = [0,-1],
               EAST  = [1,0], WEST  = [-1,0];

foreach (p in gridAIn.domain()) {
  gridB[p] = S0 * gridAIn[p] +
             S1 * (gridAIn[p + NORTH] + gridAIn[p + SOUTH] +
                   gridAIn[p + EAST ] + gridAIn[p + WEST ] );
}
```

The full MG code used for benchmarking in section 3.2 includes a 27-point stencil applied to 3D arrays. The Titanium code, like the Fortran code, uses a manually-applied stencil optimization that eliminates redundant common subexpressions, a key optimization that is further explained by Chamberlain et al [7].

3.1.3 Distributed Arrays

Titanium supports the construction of distributed array data structures using the global address space. Since distributed data structures are built from local pieces rather than declared as a distributed type, Titanium is referred to as a "local view" language by Chamberlain, Deitz, and Snyder [7]. The generality of Titanium's distributed data structures are not fully utilized in the NAS benchmarks because the data structures are simple distributed arrays. Titanium also supports trees, graphs or adaptive structures like those used by Wen and Colella [28]. Nevertheless, the general pointer-based distribution mechanism combined with the use of arbitrary base indices for arrays provides an elegant and powerful mechanism for shared data.

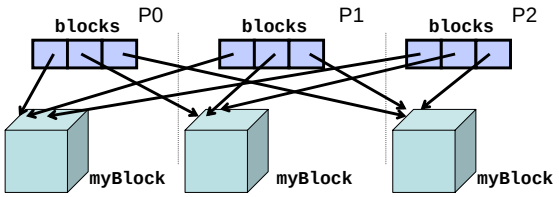


Figure 1: A distributed data structure built for MG by calling the `exchange` operation.

The following code is a portion of the parallel Titanium code for the MG benchmark. It is run on every processor and creates the `blocks` distributed array that can access any processor's portion of the grid.

```
Point<3> startCell = myBlockPos * numCellsPerBlockSide;
Point<3> endCell =
  startCell + (numCellsPerBlockSide - [1,1,1]);
double [3d] myBlock = new double[startCell:endCell];

// "blocks" is used to create "blocks3D" array
double [1d] single [3d] blocks =
  new double [0:(Ti.numProcs()-1)] single [3d];
blocks.exchange(myBlock);

// create local "blocks3D" array
// indexed by 3D block position
double [3d] single [3d] blocks3D =
  new double [[0,0,0]:numBlocksInGridSide - [1,1,1]]
  single [3d];

// map from "blocks" to "blocks3D" array
foreach (p in blocks3D.domain()) {
  blocks3D[p] = blocks[procForBlockPosition(p)];
}
```

First, each processor computes its start and end indices by performing arithmetic operations on Points. These indices are then used to create the local 3D array `myBlock`. Every processor then allocates the 1D array `blocks`, in which each element is a reference to a 3D array. The local blocks are then combined into a distributed data structure using the `exchange` keyword, which performs a gather-to-all communication that stores each processor's contribution in the corresponding element of the `blocks` array. Figure 1 illustrates the resulting data structure for an execution on three processors.

Now `blocks` is a distributed data structure, but it maps a 1D array of processors to blocks of a 3D grid. To create a more natural mapping, a 3D array called `blocks3D` is introduced. It uses `blocks` and a method called `procForBlockPosition` (not

shown) to establish an intuitive mapping from a 3D array of processor coordinates to blocks in a 3D grid. The indices for each block are in global coordinates, and these blocks are designed to overlap on some indices; these overlapped areas will serve as ghost regions.

Relative to data-parallel languages like ZPL or HPF, the "local view" approach to distributed data structures used in Titanium creates some additional bookkeeping for the programmer during data structure setup – programmers explicitly express the desired locality of data structures through allocation, in contrast with other systems where shared data is allocated with no specific affinity and the compiler or runtime system is responsible for managing the placement and locality of data. However, Titanium's pointer-based data structures are not just restricted to arrays. They can also be used to express a set of discontiguous blocks, like in AMR codes, or an arbitrary set of objects. Moreover, the ability to use a single global index space for the blocks of a distributed array means that many advantages of the global view still exist, as will be demonstrated in the next section.

3.1.4 Domain Calculus

A common operation in any grid-based code is updating ghost cells according to values stored on other processors or boundary conditions in the problem statement. An example of this is shown in figure 3. Ghost cells, a set of array elements surrounding each sub-grid, cache values belonging to the neighboring grids in the physical space of the simulation. Since adjacent sub-grids have consecutive interior cells, each sub-grid's ghost cells will overlap the neighboring grid's interior cells. Therefore, simple array operations can be used to fill in these ghost regions. Again, this migrates the tedious business of index calculations and array offsets out of the application code and into the compiler and runtime system. The entire Titanium code for updating one plane of ghost cells is as follows:

```
// use interior as in stencil code
double [3d] myBlockIn = myBlock.shrink(1);
// update overlapping ghost cells of neighboring block
blocks[neighborPos].copy(myBlockIn);
```

The array method `A.copy(B)` copies only those elements in the intersection of the index domains of the two array views in question. Using an aliased array for the interior of the locally owned block (which is also used in the local stencil computation), this code performs copy operations only on ghost values. Communication will be required on some machines, but there is no coordination for two-sided communication, and the copy from local to remote could easily be replaced by a copy from remote to local by swapping the two arrays in the copy expression.

In addition, the arbitrary index bounds of Titanium arrays allows for a global index space over the `blocks` distributed data structure. As seen, this makes it simple to select

and copy the cells in each sub-grid's ghost region. It is also used to simplify the code in the more general case of adaptive meshes.

Similar Titanium code is used for updating the other five planes of ghost cells, except in the case of the boundaries at the end of the problem domain. In those situations, the MG benchmark uses periodic boundary conditions, and an additional array view operation is required:

```
// update one neighbor's overlapping ghost cells
// across periodic boundary by logically
// shifting the local grid across the domain
blocks[neighborPos].copy(myBlockIn.translate([-256,0,0]));
```

The `translate` method translates the indices of `myBlockIn` across the problem domain, creating a new view that allows the `copy` method to correctly enforce the periodic boundary conditions.

3.1.5 Distinguishing Local Data

The `blocks` distributed array contains all the data necessary for the computation, but one of the pointers in that array references the local block which will be used for the local stencil computations and ghost cell surface updates. Titanium's global address space model allows for fine-grained implicit access to remote data, but well-tuned Titanium applications perform most of their critical path computation on data that is either local or has been copied into local memory. This avoids fine-grained communication costs which can limit scaling on distributed-memory systems with high interconnect latencies. To ensure the compiler statically recognizes the local block of data as residing locally, we declare a reference to this thread's data block using Titanium's *local* type qualifier. The original declaration of `myBlock` should have contained this local qualifier. Below we show an example of a second declaration of such a variable along with a type cast:

```
double [3d] local myBlock2 =
  (double [3d] local) blocks[Ti.thisProc()];
```

By casting the appropriate grid reference as *local*, the programmer is requesting the compiler to use more efficient native pointers to reference this array. This potentially eliminates some unnecessary overheads in array access, like dynamic checks of whether a global array access references local data. As with all type conversion in Titanium and Java, the cast is dynamically checked to maintain type safety and memory safety. However, the compiler provides a compilation mode which statically disables all the type and bounds checks required by Java semantics to save some computational overhead in production runs of debugged code.

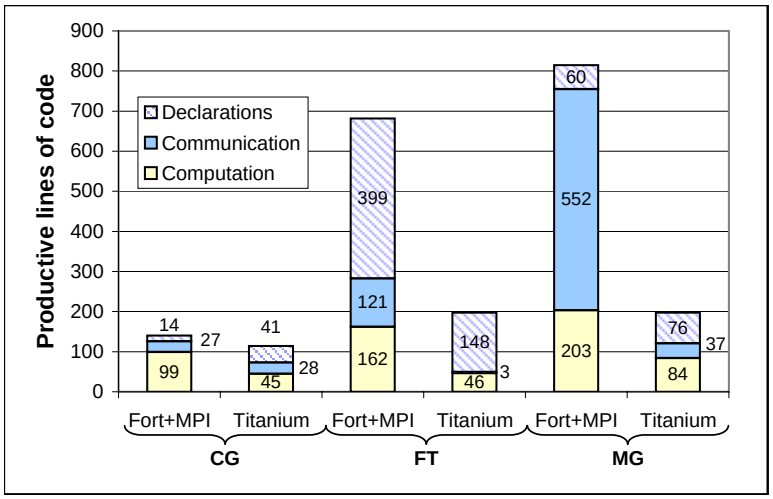


Figure 2: Timed region line count comparison

3.1.6 The MG Benchmark Implementation

The MG benchmark takes advantage of several of Titanium's most powerful features, as described in the previous sections. Almost all of the benchmark's computational methods employ stencils on a 3D mesh, which can be concisely expressed using Titanium arrays. In addition, updating the ghost cells surrounding the grid is greatly simplified by the use of Titanium's built-in domain calculus operations. Titanium's support for one-sided communication and a partitioned global address space relieves programmers from the traditionally error-prone two-sided message passing model of MPI.

Figure 2 presents a line count comparison for the Titanium and Fortran/MPI implementations of the benchmarks, breaking down the code in the timed region into categories of communication, computation and declarations. Comments, timer code, and initialization code outside the timed region are omitted from the line counts. The figure's large disparity in communication and computation line counts demonstrates that Titanium is easily more concise than Fortran/MPI for expressing the Multigrid algorithm. This productivity improvement stems from both simpler communication due to array copy operations and the leveraging of Titanium array features for local stencil computations.

While the Titanium MG code is algorithmically similar to the Fortran MG code, it is completely rewritten in the Titanium paradigm. The only real algorithmic difference is that the Fortran MG code divides work equally over all processor throughout all levels in the multigrid hierarchy, while the Titanium code performs all work on one processor below a certain multigrid level. This optimization was done to minimize small messages

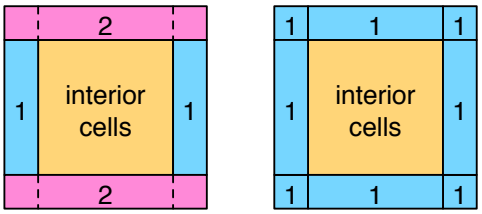


Figure 3: A 2D version of two different ways of updating ghost cells (the numbered blocks). The left figure shows two phases of communication. First, every sub-grid's two blocks labeled "1" are simultaneously populated. Then, when the blocks labeled "2" are updated, the prior phase allows them to correctly update the corner ghost cells. This corresponds to *Ti-6msg* in 3D. In the right figure, all 9 blocks are filled in simultaneously in one phase. This corresponds to *Ti-27msg* in 3D.

and reduce overall running time, but it slightly increases the Titanium line count. Such algorithmic optimizations are much simpler to implement in Titanium due to its expressive power.

3.2 Multigrid Performance

3.2.1 Titanium Hand Optimizations

Almost all of the communication in the MG benchmark involves updating ghost cells. In order to determine the fastest method for this update, two versions of the Titanium code were tested, as shown in figure 3. In the first version, all updates are performed by properly coordinating the messages passed in each dimension. Since the MG benchmark is a 3D problem, there are three phases of communication, separated by three barriers. During each of these phases, the top and bottom planes for a specific dimension are concurrently updated. Thus, in all, each processor sends up to 6 almost equally-sized messages during each update, but with three barriers for synchronization. This version is called *Ti-6msg*.

The second version, designated *Ti-26msg*, forgoes synchronization at the expense of extra messages. Therefore, there is only one communication phase followed by a single barrier. While the total communication volume is the same as *Ti-6msg*, each processor may send up to 26 messages in order to update each processor block's faces, edges, and corners. Some of these 26 messages will be very small; corner messages will only update a single double, while edge messages will update a maximum of a few hundred doubles. However, all four experimental platforms support RDMA, all messages are being sent using nonblocking array copy, and only one barrier is required. Thus, this version may do better than *Ti-6msg* on some platforms.

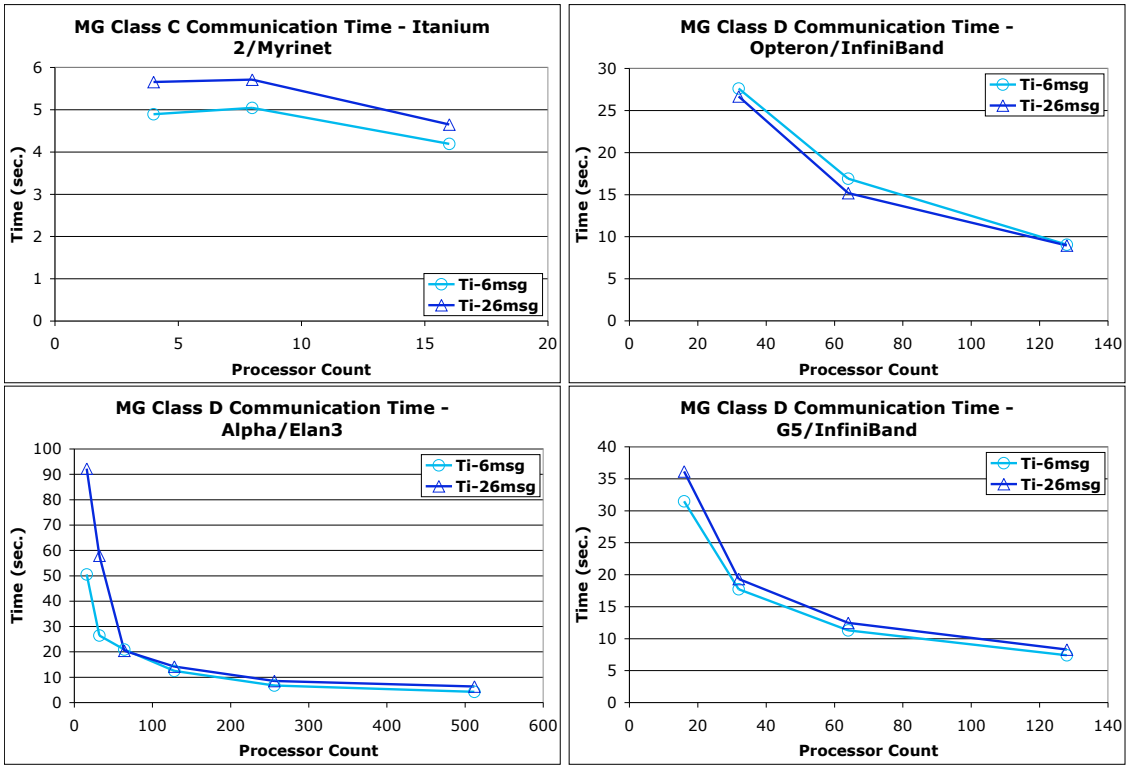


Figure 4: A comparison of the mean time (over all processors) spent updating ghost cells in MG using synchronized (*Ti-6msg*) and unsynchronized (*Ti-26msg*) communication. The timings include barrier and synchronization times for fairness.

Figure 4 shows the time spent in communication for updating ghost cells. Absolute performance can only be compared across machines for identical problem sizes and processor counts, so the smaller class C problem in the upper left should not be compared to the other three. In general, the differences in communication time between the two versions are small– the 6 message version is faster than the 26 message version on all platforms except the Opteron/Infiniband cluster. We have no clear explanation for the difference in behavior between the two Infiniband systems, since the 6 message version is faster on the G5 cluster but slower on the Opteron. However, we know that communication time is a function of the processor, compiler, and the memory system in addition to the network itself. Given the small gap between the two curves, we suspect that small differences in tuning the compiler, or low level systems software could well change the ordering of these lines.

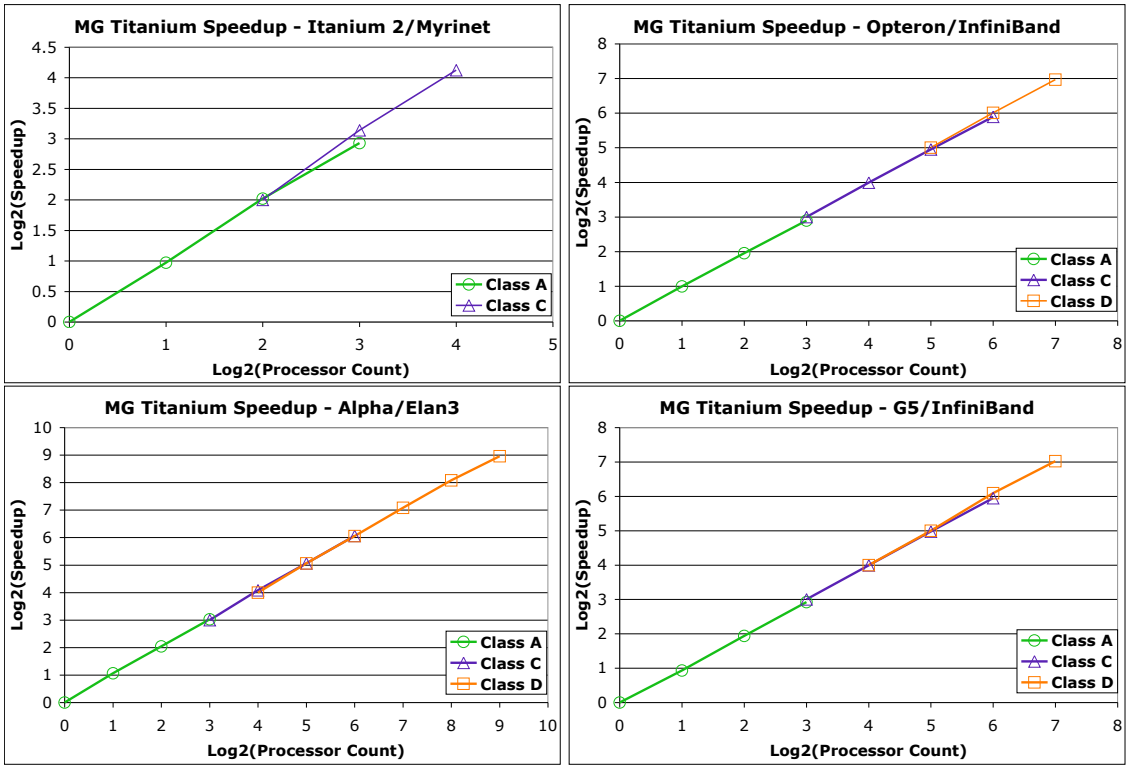


Figure 5: Log-log graphs illustrating the strong scaling of Titanium at various MG problem sizes. At the lowest processor count for a given class, the speedup is defined to be the number of processors. All speedups at higher processor counts are relative to this speedup.

3.2.2 Scalability

An important aspect of any parallel program is the ability to scale well with greater numbers of processors. In our scaling study, we chose not to use "weak scaling", a popular scaling approach that increases the problem size proportionally with the number of processors. This type of scaling would have introduced problem sizes that are not pre-defined NAS problem classes, thereby preventing us from reporting official NAS Parallel Benchmark timings. Instead, we chose to study the strong scaling (fixed problem size) of the Titanium MG benchmark for three problem classes, since a single problem size would have been impractical for the wide range of processor counts used.

Figure 5 shows the scaling results for classes A, C, and D (specified in appendix B), where the timings are the best of the two versions introduced in section 3.2.1. The baseline for each problem class is the smallest processor count, which is set to be perfect speedup. All other speedups are relative to this point.

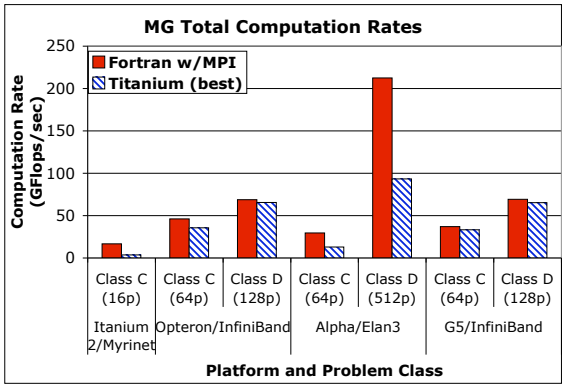


Figure 6: A comparison of MG computation rates between Fortran with MPI and Titanium. Only the largest runs for each problem size are shown.

The figure shows that the Titanium MG benchmark scales linearly over every platform, problem size, and processor count. Since the total computation is load-balanced and independent of the number of processors, the time spent in computation should scale linearly with the number of processors.

The total communication, meanwhile, is also load-balanced, but does increase with more processors. When the processor count is doubled, each processor block is halved, resulting in the total surface area increasing, but not doubling. Since ghost cells are present on the surface of each processor block, the total number of ghost cells also increases, but does not double. However, since the number of processor blocks has doubled, the number of ghost cells *per processor* actually decreases. Therefore, ignoring network contention, each processor is sending less data, so the communication time should also scale well. In actuality, network contention is also an issue, along with the fact that more processors need to complete their communication before proceeding. These effects are diminished, however, since the majority of the MG running time is spent in computation.

3.2.3 Large-Scale Performance

We have seen that the Titanium MG benchmark scales well, but we have not yet compared its performance against other languages. We finally do so in figure 6, where we compare fairly large runs of MG written in Titanium and Fortran with MPI.

The graph shows that Titanium achieves tens of gigaflops on three of the four platforms, and almost reaches 100 GFlops/sec on the Alpha. However, it does not perform as well as the Fortran code. The Fortran code performs only marginally better on the Opteron and G5 machines, but does significantly better on the Itanium and Alpha plat-

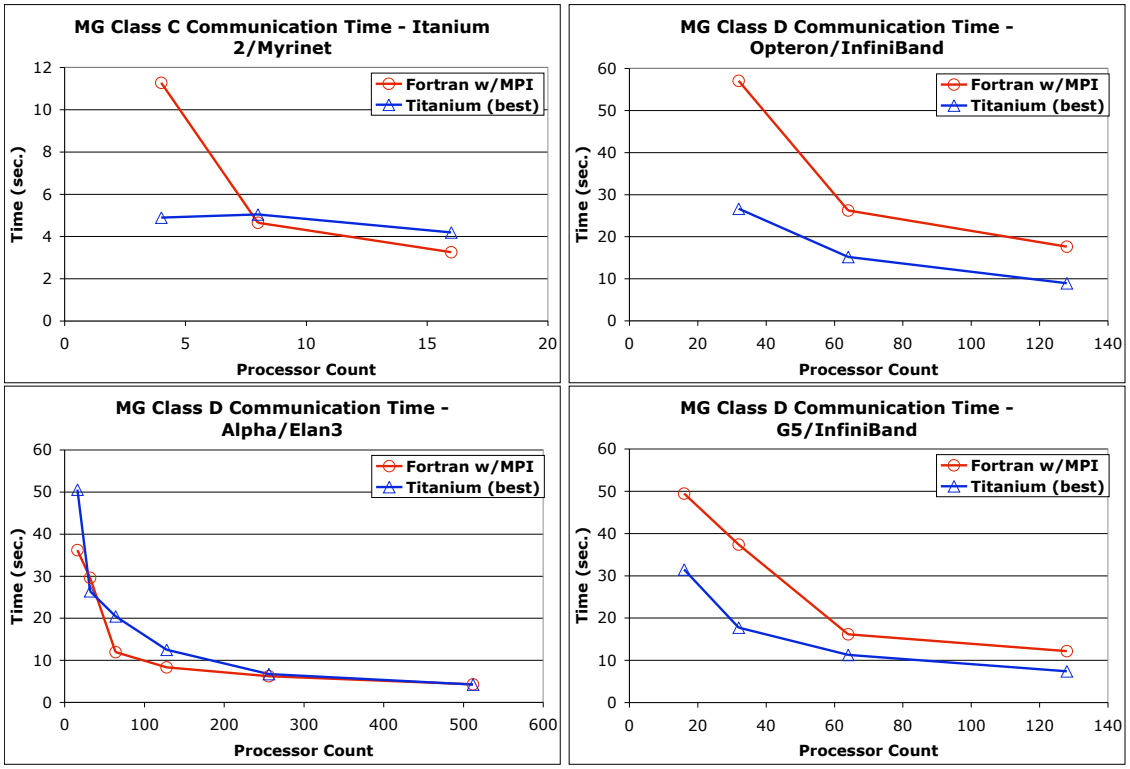


Figure 7: A comparison of the mean time (over all processors) spent updating ghost cells by Titanium and Fortran with MPI. Note that the timings include barrier and synchronization times for fairness.

forms. Whether this performance gap is due to computation, communication, or both needs to be determined.

3.2.4 Titanium vs. MPI

First, the communication times for both languages were compared, as shown in figure 7. It shows that Titanium does convincingly better on the InfiniBand machines, specifically the Opteron and the G5. However, these are also the two machines where Fortran performs slightly better overall. If Titanium is performing faster communication, it seems that the Fortran computation time is fast enough to beat Titanium overall.

On the other two machines, the Itanium and the Alpha, Titanium or Fortran does better depending on the processor count. Overall, these are the machines where Titanium does significantly worse than Fortran. Again, this would indicate that the Fortran computation is faster than that of Titanium.

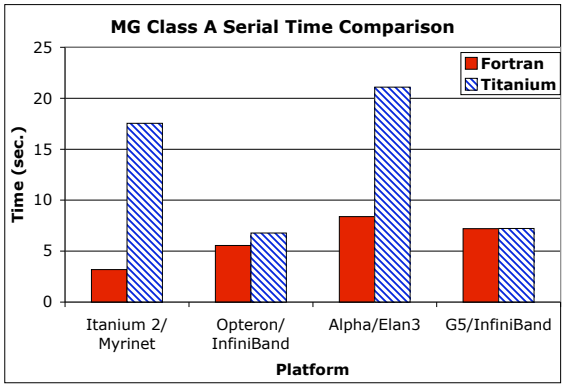


Figure 8: A comparison of serial running times between Titanium and Fortran. This is a good measure of computational speed.

A note of explanation is needed for the Alpha/Elan3 communication times. Prior benchmarks have shown that Titanium does better than MPI on this network, but figure 7 seems to contradict this for some processor counts. In actuality, there is greater variability in the Titanium communication timings for this platform, and since the mean times are displayed, the mean Fortran timings sometimes do better. The reason for this variability in Titanium's communication time is unclear. The MG communication is load-balanced, but network contention, among a host of other factors, could play a role.

3.2.5 Serial Performance

The previous section suggests that computation in Titanium is significantly slower than Fortran on the Itanium and Alpha machines, and may be marginally slower on the Opteron and G5 machines. To determine if this is so, a comparison was done of MG serial running times in Titanium and Fortran. By eliminating any network communication, this comparison should give a good sense of how fast computation is being performed in both languages. As seen in figure 8, the prior section's predictions are confirmed.

There are two possible reasons for this serial performance gap. First, since the Titanium compiler produces C code, the C compiler may lose performance from translating mechanically generated source for which it is not tuned. Second, the Fortran compiler may generate better machine code than the C compiler, especially on scientific codes.

4 Fourier Transform (FT)

The FT benchmark uses 3D Fast Fourier Transforms (FFTs) to solve a partial differential equation over several time steps. Each 3D FFT is actually performed as a series 1D FFTs across each dimension of a regular 3D grid. Therefore, we parallelize the problem by partitioning the grid into slabs; this allows each slab to execute 1D FFTs in two of the dimensions locally. However, in order to perform the final 1D FFT, an all-to-all transpose is required.

The Titanium FT code calls the FFTW library [13] for the local 1D FFTs, so for the sake of fairness, the Fortran FT code was modified to do the same. Both implementations follow the same general algorithm; they both perform two local 1D FFTs, followed by a global transpose, and finally another local 1D transform. However, there are algorithmic differences between the two implementations that were not changed. The Titanium algorithm takes full advantage of nonblocking array copy in the 3D FFT by overlapping computation with communication. On the other hand, the Fortran/MPI 3D FFT is bulk synchronous, so the computation and communication each occur in different phases. Despite these differences, we believe that the performance comparison is a fair one, since the original NAS Benchmarks paper [1] states that "the choice of data structures, algorithms, processor allocation, and memory usage are all (to the extent allowed by the specification) left open to the discretion of the implementer". The paper also justifies Titanium's use of nonblocking array copy by asserting that "programmers are free to utilize language constructs that give the best performance possible on the particular system being studied."

Although both languages call the FFTW library for the 1D FFTs, the strides involved in the two versions differ. The Fortran version only executes unit-to-unit 1D FFTs, meaning that a local transpose is required. In contrast, the Titanium version executes two strided FFTs, one of which helps to avoid the local transpose, as well as a unit-to-unit transform.

The communication pattern is also different between the two versions. Titanium's FT implementation optimizes the all-to-all transpose by taking full advantage of the one-sided communication model [2], while Fortran again uses MPI's two-sided messaging. The specifics involved are discussed in sections 4.2.1 and 4.2.4.

4.1 Titanium Features in the Fourier Transform Benchmark

The FT benchmark illustrates several useful Titanium features, both for readability and performance. Since it is heavily dependent on computation with Complex numbers, the Complex class was declared to be *immutable* for efficiency. To make the code using the Complex class more readable, several methods employ *operator overloading*. To access specialized libraries, the Titanium code uses *cross-language calls*. Finally, *nonblocking*

Java Version

```
public class Complex {
  private double real, imag;
  public Complex(double r, double i)
    { real = r; imag = i; }
  public Complex add(Complex c)
    { ... }
  public Complex multiply(double d)
    { ... }
  ...
}
   /* sample usage */
Complex c = new Complex(7.1, 4.3);
Complex c2 = c.add(c).multiply(14.7);
```

Titanium Version

```
public immutable class Complex {
  public double real, imag;
  public Complex(double r, double i)
    { real = r; imag = i; }
  public Complex op+(Complex c)
    { ... }
  public Complex op*(double d)
    { ... }
  ...
}
   /* sample usage */
Complex c = new Complex(7.1, 4.3);
Complex c2 = (c + c) * 14.7;
```

Figure 9: Complex numbers in Java and Titanium

array copy increased the speed of data transfer.

4.1.1 Immutables and Operator Overloading

The Titanium *immutable* class feature provides language support for defining application-specific primitive types (often called "lightweight" or "value" classes). These types allow for the creation of user-defined unboxed objects that are passed directly, analagous to C structs. As a result, they avoid the pointer-chasing overheads and allocation that would otherwise be associated with the use of tiny objects in Java.

One compelling example of the use of immutables is for defining a Complex number class, which is used to represent the complex values in the FT benchmark. Figure 9 compares how one might define a Complex number class using either standard Java Objects or Titanium immutables.

In the Java version, each complex number is represented by an Object with two fields corresponding to the real and imaginary components, and methods provide access to the components and mathematical operations on Complex objects. If one were then to define an array of such Complex objects, the resulting in-memory representation would be an array of pointers to tiny objects, each containing the real and imaginary components for one complex number. This representation is wasteful of storage space; by imposing the overhead of storing a pointer and an Object header for each complex number, the required storage space can easily be doubled. More importantly for scientific computing purposes, such a representation induces poor memory locality and cache behavior for operations over large arrays of such objects. Finally, compared to standard mathematical notation, the method-call syntax required for performing operations on the Complex Objects in standard Java is clumsy.

Titanium allows easy resolution of the listed performance issues by adding the *immutable* keyword to the class declaration, as shown in the figure. This one-word change declares the Complex type to be a value class, which is stored as an unboxed type in the containing context (e.g. on the stack, in an array, or as a field of a larger object). The figure illustrates the framework for a Titanium-based implementation of Complex using immutables and operator overloading, which mirrors the implementation provided in the Titanium standard library (ti.lang.Complex) that is used in the FT benchmark.

Immutable types are not subclasses of java.lang.Object, and induce no overheads for pointers or Object headers. They are also implicitly final, which means they never pay execution-time overheads for dynamic method-call dispatch. An array of Complex immutables is represented in memory as a single contiguous piece of storage containing all the real and imaginary components, with no pointers or Object overheads. This representation is significantly more compact in storage and efficient in runtime for computationally-intensive algorithms such as FFT.

The figure also demonstrates the use of Titanium's operator overloading, which allows one to define methods corresponding to the syntactic arithmetic operators applied to user classes (the feature is available for any class type, not just immutables). This allows a more natural use of the $+$ and $*$ operators to perform arithmetic on the Complex instances, allowing the client of the Complex class to handle the complex numbers as if they were built-in primitive types.

4.1.2 Cross-Language Calls

Titanium allows the programmer to make calls to kernels and libraries written in other languages, enabling code reuse and mixed-language applications. This feature allows programmers to take advantage of tested, highly-tuned libraries, and encourages shorter, cleaner, and more modular code. Several of the major Titanium applications make use of this feature to access computational kernels such as vendor-tuned BLAS libraries.

Titanium is implemented as a source-to-source compiler to C, which means that any library offering a C interface is potentially callable from Titanium. Because Titanium has no JVM, there is no need for a complicated calling convention (such as the Java JNI interface) to preserve memory safety. [1] To perform cross-language integration, programmers simply declare methods using the *native* keyword, and then supply implementations written in C.

The Titanium NAS FT implementation featured in this paper calls the FFTW [13]

[1] The Berkeley Titanium compiler uses the Boehm-Weiser conservative garbage collector [4] for automatic memory management, eliminating the need to statically identify the location of all pointers at runtime, at a small cost in collector precision and performance relative to copying garbage collectors that are typically used by standard Java implementations.

library to perform the local 1D FFT computations, thereby leveraging the auto-tuning features and machine-specific optimizations of its FFT kernel implementation. Although the FFTW library does offer a 3D MPI-based parallel FFT solver, our benchmark only uses the serial 1D FFT kernel; Titanium code is used to create and initialize all the data structures, as well as to orchestrate and perform all the interprocessor communication operations.

One of the challenges of the native code integration with FFTW was manipulating the 3D Titanium arrays from within native methods, where their representation as 1D C arrays is exposed to the native C code. This was a bit cumbersome, especially since the FT implementation intentionally includes padding in each row of the array to avoid cache-thrashing. However, it was only because of Titanium's support for true multidimensional arrays that such a library call was even possible, since the 3D array data is stored natively in a row-major, contiguous layout. In contrast, Java's layout of "multidimensional" arrays is as 1D arrays of pointers to 1D arrays. Such a layout is likely not contiguous, making such library calls all but impossible.

4.1.3 Nonblocking Array Copy

Titanium's explicitly nonblocking array copy library methods helped in implementing a more efficient 3D FFT, as shown for a three-processor execution in figure 10.

The Fortran code performs a bulk-synchronous 3D FFT, whereby each processor performs two local 1D FFTs, then all the processors collectively perform an all-to-all communication, followed by another local 1D FFT. This algorithm has two major performance flaws. First, because each phase is distinct, there is no resulting overlap of computation and communication - while the communication is proceeding, the floating point units on the host CPUs sit idle, and during the computation the network hardware is idle. Secondly, since all the processors send messages to all the other processors during the global transpose, the interconnect can easily get congested and saturate at the bisection bandwidth of the network (which is often significantly less than the aggregate node bandwidth in large-scale cluster systems). This can result in a much slower communication phase than if the same volume of communication were spread out over time during the other phases of the algorithm.

Both these issues can be resolved with a slight reorganization of the 3D FFT algorithm employing nonblocking array copy. The new algorithm, implemented in Titanium, first performs a local strided 1D FFT, followed by a local unit-strided 1D FFT. During this unit-strided 1D FFT, a processor sends a message as soon as the remote processor's portion of the current local plane is computed. By staggering the messages throughout the computation, the network is less likely to become congested and is more effectively

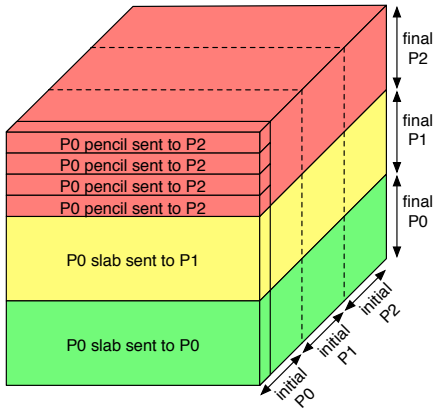


Figure 10: Illustration of the FFT all-to-all transpose, where the dotted lines indicate the original layout of the grid. The Fortran code completes the FFT computation in a processor slab before starting the global transpose. In contrast, the Titanium code sends either slabs or pencils (but not both) as soon as they are computed. The diagram shows the slabs/pencils that constitute the first plane of P0's initial slab. By sending these slabs/pencils using nonblocking array copy, Titanium can overlap computation and communication.

utilized.

Moreover, we send these slabs using nonblocking array copy, addressing the other issue with the original algorithm. Nonblocking array copy allows us to inject the message into the network and then continue with the local FFTs, thus overlapping most of the communication costs incurred by the global transpose with the computation of the second FFT pass. When correctly tuned, nearly all of the communication time can be hidden behind the local computation. The only communication costs that can never be hidden through overlap are the software overheads for initiating and completing the non-blocking operations. Our GASNet communication system has been specifically tuned to reduce these host CPU overheads to the bare minimum, thereby enabling effective overlap optimizations such as those described here. Reorganizing the communication in FT to maximize overlap results in a large performance gain, as seen in figure 11.

4.1.4 The FT Benchmark Implementation

Figure 2 shows that the Titanium implementation of FT is considerably more compact than the Fortran/MPI version. There are three main reasons for this. First, over half the declarations in both versions are dedicated to verifying the checksum, a Complex number that represents the correct "answer" after each iteration. The Titanium code does this a

bit more efficiently, thus saving a few lines. Secondly, the Fortran code performs cache blocking for the FFTs and the transposes, meaning that it performs them in discrete chunks in order to improve locality on cache-based systems. Moreover, in order to perform the 1D FFTs, these blocks are copied to and from a separate workspace where the FFT is performed. While this eliminates the need for extra arrays for each 1D FFT, any performance benefit hinges on how quickly the copies to and from the workspace are done. The Titanium code, on the other hand, allocates several arrays for the 3D FFT, and therefore does not do extra copying. It is consequently shorter code as well. Finally, Titanium's domain calculus operations allow the transposes to be written much more concisely than in Fortran, resulting in a 121 to 3 disparity in communication line counts.

4.2 Fourier Transform Performance

4.2.1 Titanium Hand Optimizations

Similarly to MG, we again try to reduce the communication time, now by optimizing the FFT all-to-all transpose shown in figure 10. In the baseline version of the Titanium code, designated *Ti-bl_slabs*, each processor sends a slab as soon as it is computed using a blocking array copy operation. Since the communication blocks, it does not overlap with the computation.

The first optimization is to make the array copy operation nonblocking. This version, called *Ti-nbl_slabs*, allows the processor to inject a processor slab into the network and immediately start computing the next slab, thereby allowing communication-computation overlap.

The second optimization, called *Ti-nbl_pencils*, goes one step further. Instead of sending slabs, each pencil within each slab is sent using nonblocking array copy as soon as it is computed. Since smaller messages are being sent more rapidly, network bandwidth should be better utilized. However, too many network messages may also have the opposite effect of inducing network congestion.

Figure 11 shows the time spent in communication for the FFT global transpose. In order to do a fair comparison, the measured time includes barrier and synchronization times as well. As expected, we see that *Ti-nbl_slabs* always does better than *Ti-bl_slabs*. The performance of *Ti-nbl_pencils* is more erratic, however. Due to the sheer number of messages being injected into the portion of the network connecting the running nodes, there were some problems running this code at small processor counts. Occasionally the network would deadlock and no timing results were returned. At other times, the network would become congested and cause the timings to be unexpectedly long. But at higher processor counts, *Ti-nbl_pencils* performs comparably to *Ti-nbl_slabs*.

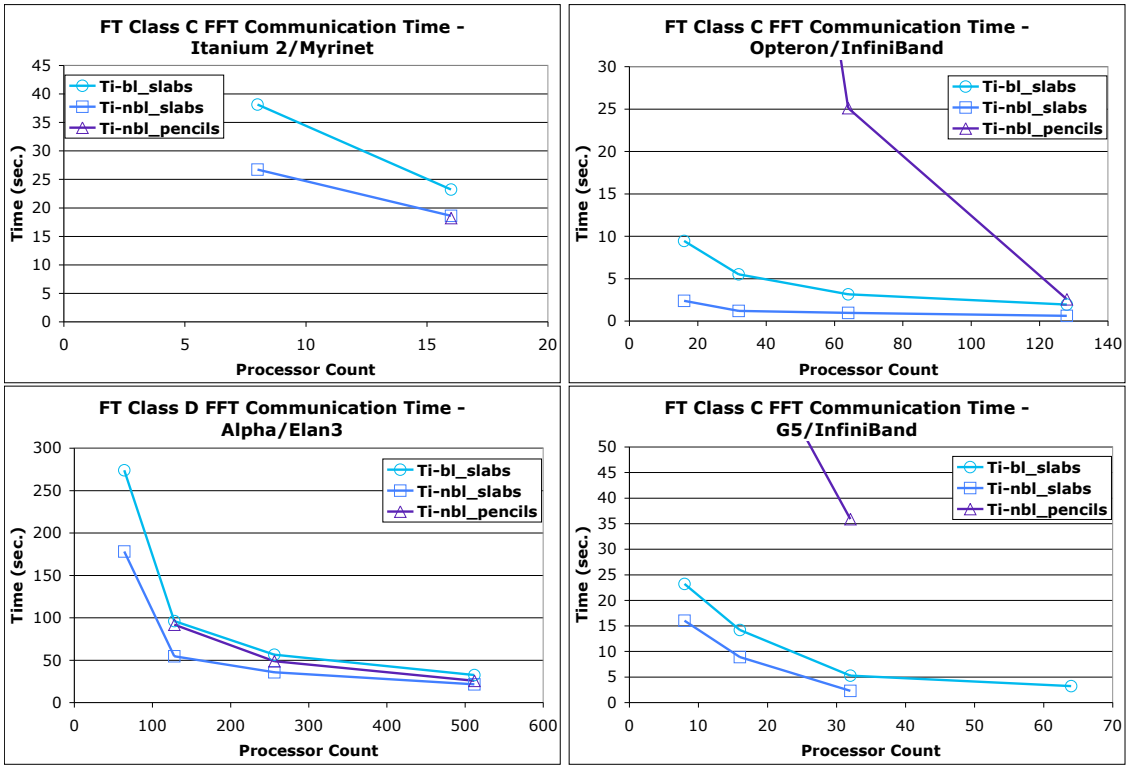


Figure 11: A comparison of the mean time (over all processors) spent in the FFT all-to-all transpose. Version *Ti-bl_slabs* sends slabs using blocking array copy, *Ti-nbl_slabs* sends slabs using nonblocking array copy, and *Ti-nbl_pencils* sends nonblocking pencils. The timings include barrier and synchronization times for fairness.

A note of explanation is needed for the G5/InfiniBand machine. Due to network layer issues, there was some difficulty running the FT Class C problem at 64 processors. As a result, only the *Ti-bl_slabs* is shown for that particular case.

4.2.2 Scalability

To see how well the best Titanium version of the FT benchmark scales, we again examine log-log speedup graphs, as shown in figure 12. It shows that at small scales (class A), FT scales sublinearly on the Itanium and Alpha machines. The class A problem does scale linearly on the Opteron, however, and is superlinear on the G5. At larger scales (class C and D), FT scales linearly (and sometimes superlinearly) over all platforms.

In order to explain these effects, we need to look at the computation and the com-

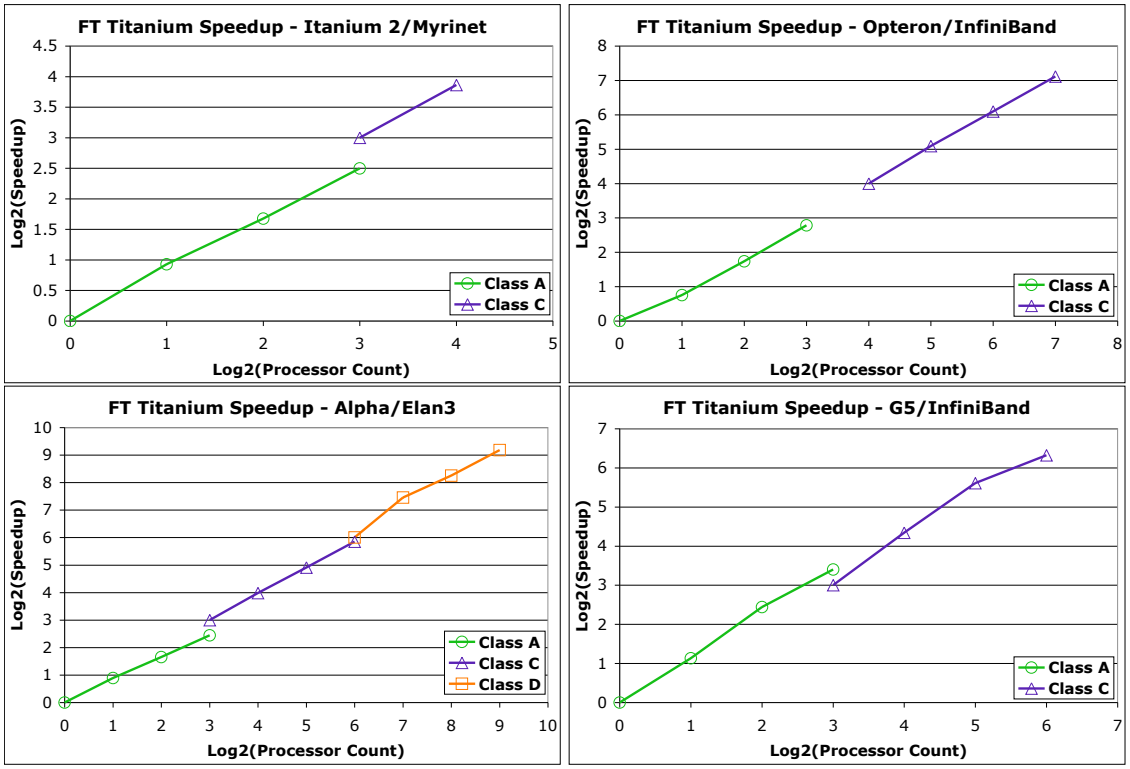


Figure 12: Log-log graphs illustrating the strong scaling of Titanium at various FT problem sizes. At the lowest processor count for a given class, the speedup is defined to be the number of processors. All speedups at higher processor counts are relative to this speedup.

munication separately. The computation is load-balanced, and the total computation is independent of the processor count (in this case). By doubling the number of processors, each processor slab is divided in two. When sending slabs, this may speed up one of the strided 1D FFTs because the stride through memory is shorter. Ignoring this effect, the computation should scale linearly.

The communication is also load-balanced, and the total communication volume is also the same regardless of the number of processors. Since we are using only one processor per node, when the processor count is doubled, the number of running nodes in the network doubles as well. This allows us to spread the communication volume over more links in the network, reducing the probability of congestion. In addition, each processor now sends half as much data, so ignoring network congestion, all blocking communication should also scale linearly. However, section 4.2.1 showed that the best Titanium version usually sends nonblocking slabs, so much of the communication time is overlapped with

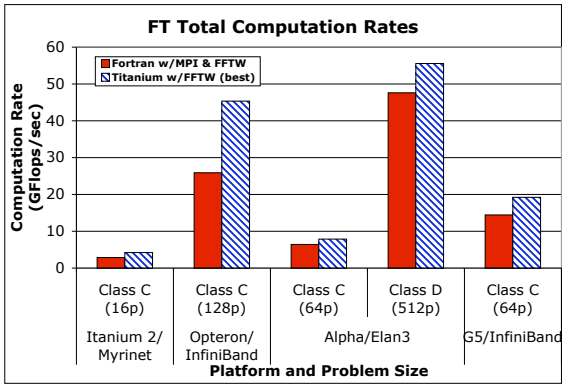


Figure 13: A comparison of FT computation rates between Fortran with MPI and Titanium. Only the largest runs for each problem size are shown.

computation. This reduces any effects from communication scaling.

4.2.3 Large-Scale Performance

Now we compare Titanium's performance to that of Fortran. There is some non-FFT computation in the FT benchmark, but it is minimal. Consequently, since FFTW is being used for both Titanium and Fortran, any performance disparities should largely be attributable to communication.

Figure 13 shows that the best Titanium version does better than Fortran for large scales (classes C and D) on all platforms. Since the best Titanium version is usually nonblocking slabs, hiding much of the global communication time within computation helps Titanium significantly outperform the Fortran/MPI code.

4.2.4 Titanium vs. MPI

To confirm that communication is the reason for performance gap between Titanium and Fortran, only the communication times are compared in figure 14. It confirms that Titanium's one-sided communication performs much better than MPI's send-and-receive paradigm for the NAS FT code.

5 Conjugate Gradient (CG)

The CG benchmark uses the conjugate gradient method to iteratively solve for the smallest eigenvalue of a large, sparse symmetric positive-definite matrix. Therefore, the main data

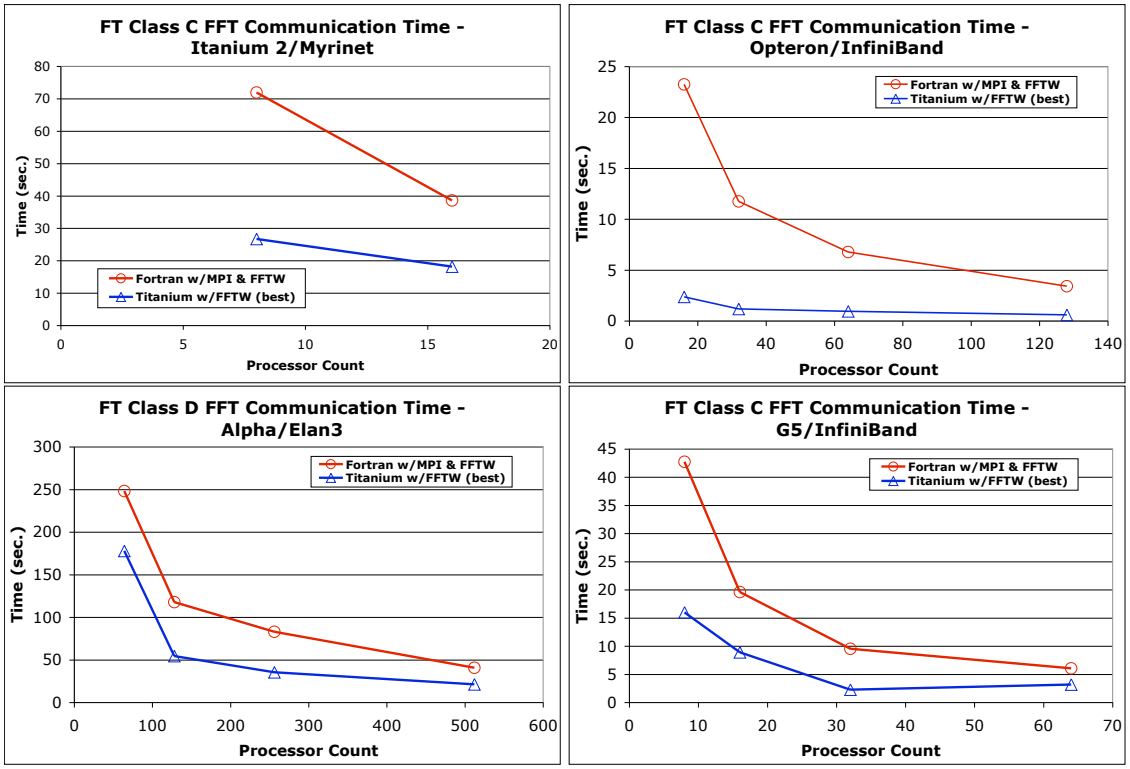


Figure 14: A comparison of the mean time (over all processors) spent in the FFT all-to-all transpose by Titanium and Fortran with MPI. The timings include barrier and synchronization times for fairness.

structure is a sparse matrix in Compressed Sparse Row (CSR) format, as discussed in section 5.1.1. Almost all of the benchmark's computations derive from sparse matrix-vector multiply (SpMV), which requires an indirect array access for each matrix nonzero.

The problem is parallelized by blocking the matrix in both dimensions. Specifically, the matrix's columns are halved, followed by the rows, and then repeating until there a block for every processor. Therefore, the number of processor columns will either be double or equal to the number of processor rows. Figure 15 illustrates how the matrix is distributed over 4 and 8 processors. In general, since the matrix is sparse, each processor will receive an unequal number of nonzeros. As a result, the problem is not perfectly load-balanced like MG and FT. However, since the matrix is relatively uniform, the problem is not severely load-imbalanced either. Each processor also receives the sections of the source and destination vectors that are needed for the local SpMV.

This type of partitioning necessitates that every processor sum together all of the des-

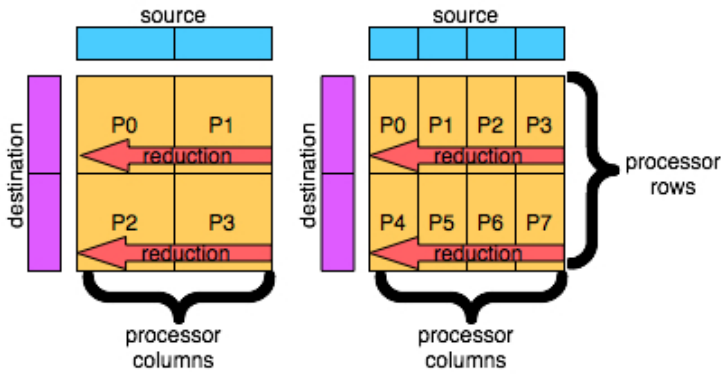


Figure 15: The parallel partitioning of CG over 4 and 8 processors. Note that each reduction in the 4-processor case is over 2 processors, while in the 8-processor case each reduction is over 4 processors.

tination vectors in its processor row. This reduction is efficiently implemented in NAS CG by using a dispersion algorithm requiring $\log_2 c$ phases, where c is the number of processor columns. The synchronization of this algorithm in Titanium is discussed in section 5.2.1.

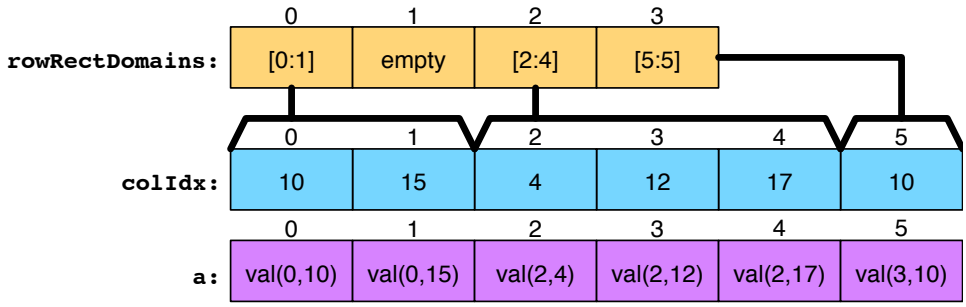
5.1 Titanium Features in the Conjugate Gradient Benchmark

The following sections highlight the Titanium features used to implement the CG benchmark that have not been discussed in prior sections. We focus primarily on Titanium foreach loops and the implementation of pairwise synchronization for producer-consumer communication.

5.1.1 Foreach Loops

As described in section 3.1.2, Titanium has an unordered loop construct called *foreach* that simplifies iteration over multidimensional arrays and provides performance benefits. If the order of loop execution is irrelevant to a computation, then using a *foreach* loop to traverse the points in a *RectDomain* explicitly allows the compiler to reorder loop iterations to maximize performance– for example, by performing the automatic cache blocking and tiling optimizations described by Pike [24, 25]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

Another example of the use of the foreach loop can be found in the sparse matrix-vector multiplies performed in every iteration of the CG benchmark. The sparse matrix below is stored in CSR format, with data structures illustrated in Figure 16. The




```
// the following represents a matrix in CSR format
// all three arrays were previously populated
RectDomain<1> [1d] rowRectDomains; // RectDomains of row indices
int [1d] colIdx;  // column index of nonzeros
double [1d] a;    // nonzero matrix values
...
public void multiply(double [1d] sourceVec, double [1d] destVec) {
  foreach (i in rowRectDomains.domain()) {
    double sum = 0;
    foreach (j in rowRectDomains[i])
      sum += a[j] * sourceVec[colIdx[j]];
    destVec[i] = sum;
} }
```

Figure 16: The Compressed Sparse Row (CSR) format for representing a sparse matrix, followed by Titanium code for sparse matrix-vector multiply.

`rowRectDomains` array contains a RectDomain for each row of the matrix. Each RectDomain contains its row's first and last indices for arrays `colIdx` and `a`.

In the figure, the logical sparse array has four rows (one of which is entirely zero) containing a total of six nonzero data values. The nonzero values are stored contiguously in the array `a`, while arrays `rowRectDomains` and `colIdx` hold the metadata used to perform sparse indexing. More specifically, the index of the `rowRectDomains` array is the row number and the value of the `colIdx` array is the column number of each nonzero value in array `a`.

This calculation uses nested foreach loops that highlight the semantics of foreach; namely, that the loop executes the iterations serially in an unspecified order. The outer loop is expressed as a foreach because each of the dot products operates on disjoint data, so ordering does not affect the result. The inner loop is also a foreach, which indicates that the sum can be done in any order. This allows the compiler to apply associativity and commutativity transformations on the summation. Although these transformations may affect the exact result, they do not affect algorithmic correctness for reasonable matrices.

5.1.2 Point-to-point Synchronization

Both the Titanium and Fortran NAS CG implementations use a matrix that is partitioned in both dimensions. Portions of both source and destination vectors are replicated on the processors that use them, so the only communication in the sparse matrix-vector product involves reductions over each processor row (a *team reduction*) to perform a dot product. The code performs this reduction using a pairwise exchange algorithm with $\log_2 c$ phases, where c is the number of processor columns, and synchronization between processor pairs is required in each phase as data is passed. An example of the CG layout is shown in figure 15.

Implementations using MPI for communication usually rely on the synchronization provided by the messaging handshake, as all two-sided message-passing operations imply both a data transfer and synchronization between sender and receiver. One-sided communication decouples data transfer from synchronization, because there is no receive operation and generally no explicit action or notification at the remote process. Consequently, some additional action is required to achieve pairwise synchronization between producers and consumers of a data stream in algorithms such as the CG reduction. This synchronization can naively be achieved using barriers. However, as shown in section 5.2.1, a faster method is to employ more direct and efficient techniques for point-to-point synchronization between pairs of processors. This eliminates the over-synchronization overheads imposed by using global barriers for pairwise synchronization.

Our implementation of the CG benchmark uses in-memory flags to perform pairwise synchronization between producers and consumers during the team reduction steps of the algorithm. Each producer-to-consumer communication is achieved using a two-step communication based on standard in-memory signaling algorithms, extended to the global address space. In the first step, the producer pushes the data from the local source memory to a prearranged destination area on the remote consumer, using a Titanium array copy operation that expands to an RDMA put operation on GASNet backends where that functionality is available on the network hardware. When the blocking array copy returns to the caller, this indicates remote completion of the data transfer operation (which in the case of a cluster network is usually detected via link-level acknowledgment of the RDMA put operation from the NIC hardware at the remote target). Once the data transfer is complete, the producer writes a flag in an array on the remote consumer, initiating a single word put operation notifying the consumer that data is available. Once the consumer is ready to consume data, it spins waiting for the flag value to become non-zero and then consumes the data which is waiting in the prearranged location.

This technique decouples data transfer from synchronization, thereby achieving the required point-to-point synchronizing data transfer. However, it is worth noting the oper-

ation is still one-sided in flavor– specifically, the initiator provides complete information about the locations and sizes of all data transfers, and no explicit action is required at the target to complete the data transfers. As a result, this method of communication can still reap the performance benefits of fully one-sided communication, namely the use of zero-copy transfers with no rendezvous messaging delays or eager buffering costs.

The algorithm described above is quite efficient on systems with hardware support for shared memory, such as SMP's and DSM systems such as the SGI Altix or Cray X-1. However, the algorithm is less efficient for cluster networks, because the initiator waits for the completion of the data transfer before issuing the flag write. Therefore, the one-way latency for the synchronizing data transfer on a distributed-memory network amounts to roughly one and a half round-trips on the underlying network. Explicitly nonblocking communication can be used in some algorithms to overlap most of this latency with independent computation and other data transfers. In the specific case of CG's pairwise-exchange reduction, however, there is no other independent work available for overlap. We are exploring ways of providing a fast point-to-point synchronization mechanism that would avoid the round-trip latency (building on existing runtime support) as well as generalizing and improving the collective communication library so it could be used in the parallel dot product on a subset of processors.

5.1.3 The CG Benchmark Implementation

The CG benchmark demonstrates the utility of Titanium arrays and foreach loops in implementing the data structures and computations for sparse matrices stored in CSR format. In addition, the team reductions required by the 2D blocked decomposition motivate the use of point-to-point synchronization constructs to achieve efficient pairwise synchronization in producer-consumer communication patterns. This is a relatively new area of exploration in the context of PGAS languages like Titanium where all communication is fully one-sided and implicit.

Figure 2 illustrates the line count comparison for the timed region of the Fortran/MPI and Titanium implementations of the CG benchmark. In contrast with MG, the amount of code required to implement the timed region of CG in Fortran/MPI is relatively modest, primarily owing to the fact that no application-level packing is required or possible for this communication pattern. In addition, MPI's message passing semantics implicitly provide pairwise synchronization between message producers and consumers, so no additional code is required to achieve that synchronization.

The Titanium implementation is comparably concise. Despite the extra code required to achieve pairwise synchronization, the amount of communication code is roughly equivalent. The computation code is 54 lines shorter in Titanium, but there are 27 additional

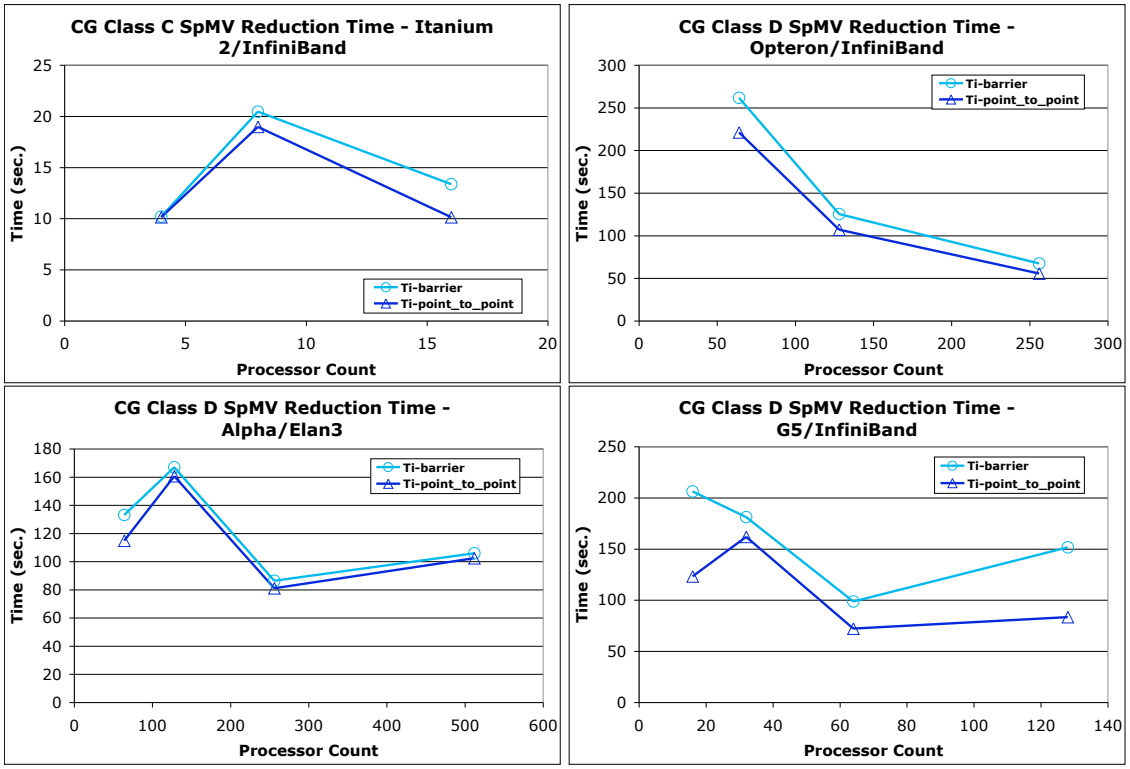


Figure 17: A comparison of the mean time (over all processors) spent in SpMV reduction using barriers (*Ti-barrier*) and point-to-point synchronization (*Ti-point_to_point*).

declaration lines. However, this difference is due to the use of object-orientation and modularity in implementing the vector operations as methods in a Titanium Vector class. In the Fortran code, those same operations are implemented by expanding the vector operations inline as many very similar do loops, resulting in less readability.

5.2 Conjugate Gradient Performance

5.2.1 Titanium Hand Optimizations

As mentioned in section 5, the CG SpMV reduction is performed as a dispersion algorithm requiring $\log_2 c$ phases, where c is the number of processor columns. If r is the number of processor rows, we know that $c = r$ when the processor count is square, and $c = 2r$ when it is not (as shown in figure 15). Therefore, when we double a square processor count (e.g., go from 4 to 8 or from 16 to 32), we add an extra dispersion phase to the reduction.

In addition, an independent reduction is being performed in each processor row. Thus,

when we double a non-square processor count (e.g., go from 8 to 16 or from 32 to 64), we introduce another processor row. However, since all processor rows perform an equal amount of work, an extra processor row lessens the size of the reduction in the other processor rows.

In order to minimize the time Titanium spent in communication, two different ways of implementing the SpMV reduction were compared. The first method uses barriers to synchronize after each dispersion phase, while the second method uses point-to-point communication. The results, as shown in figure 17, indicate that point-to-point synchronization does better on every platform and processor count, although the timings are sometimes very close. This is not an unexpected result, as discussed in section 5.1.2. The barrier implementation does a global synchronization after each dispersion phase, when all that is required is synchronization for each processor row. The point-to-point version, in general, is a better solution because it only synchronizes between the sending and receiving processors.

Another notable feature of the graphs is that instead of a smooth line or curve, multiple peaks are present. Much of this can be attributed to the parallel partitioning of the problem. In general, the square processor counts perform faster reductions than the next higher processor count because they perform one fewer phase in the dispersion algorithm.

5.2.2 Scalability

Figure 18 displays how the best Titanium CG code scales at different sizes. Since the point-to-point code in the previous section always performed better, this is the code for which the results are shown. We see that, in general, Titanium scales well over all platforms and processor counts.

Since SpMV takes the great majority of the time spent in CG, we examine the SpMV computation and communication phases separately in order to understand the speedup graphs. The time spent in SpMV computation dominates the reduction time, so scaling the computation is more important in this case.

The sparse matrix is fairly uniform, so partitioning it does not introduce much load imbalance. In addition, the total computation volume is independent of processor count, so partitioning the matrix does not introduce extra computation either. In fact, due to cache effects, the partitioning actually increases the computational rate per processor. For larger processor counts, each processor receives a smaller matrix block that is more likely to fit into cache, thereby allowing for faster computational rates. For example, for the Class C problem on the Alpha, going from 32 to 64 processors increases the average per processor SpMV computation rate from 163 MFlops/sec to 224 MFlops/sec. The increase is not typically this large, but the computation time does scale superlinearly.

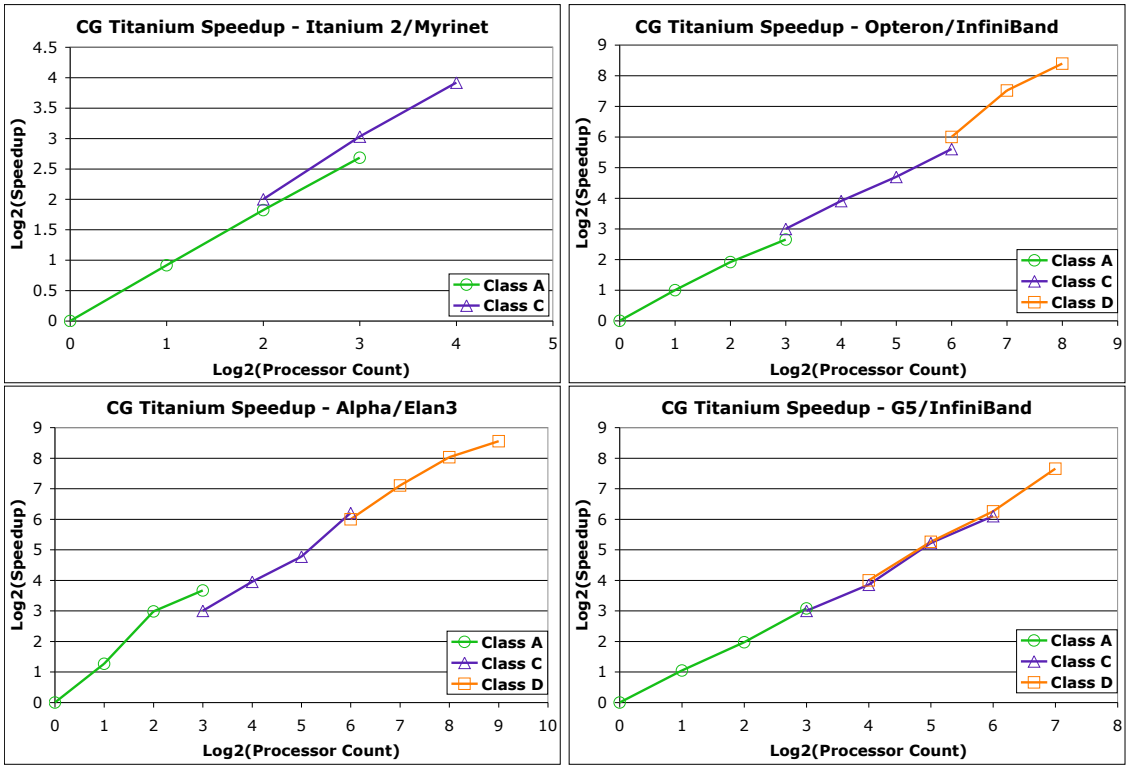


Figure 18: Log-log graphs illustrating the strong scaling of Titanium at various CG problem sizes. At the lowest processor count for a given class, the speedup is defined to be the number of processors. All speedups at higher processor counts are relative to this speedup.

However, the total communication involved in the SpMV reduction, while load-balanced, does increase with processor count. Introducing an extra processor row does not increase the total communication, but adding a processor column does, since a new phase is added to the dispersion algorithm. In general, the communication volume *per processor* decreases with increased processor count, but the extra phases added to the dispersion algorithm slow the reduction time significantly. As a result, the time spent in the reduction should not be expected to scale linearly. But, since the reduction time is less than the time in SpMV computation, these effects are diminished, and the overall speedup is linear.

5.2.3 Large-Scale Performance

Figure 19 compares the performance of both Titanium and Fortran at the largest processor counts for each problem size. It shows that Titanium's performance trails that of Fortran

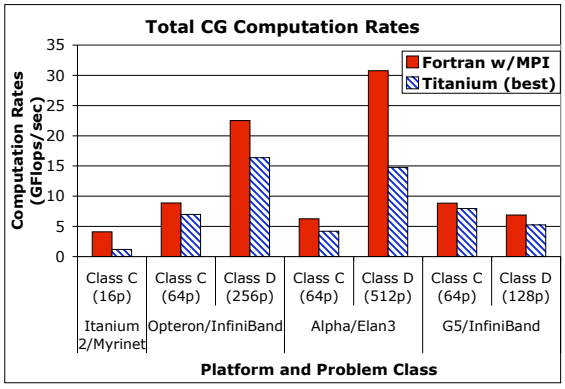


Figure 19: A comparison of CG computation rates between Fortran with MPI and Titanium. Only the largest runs for each problem size are shown.

in every case. The reasons for this gap are analyzed in the next two sections.

5.2.4 Titanium vs. MPI

First, we examine the time spent in the SpMV reduction, as shown in figure 20. It again shows the same peaks in the Titanium performance that we saw in section 5.2.2. More importantly, though, it shows that outside the Itanium platform, the Titanium performance is worse than MPI. As fully explained in section 5.1.2, this is due to the way point-to-point synchronization is implemented in the one-sided communication paradigm. Titanium requires 1.5 round trips for each phase of the dispersion– first it does a data transfer, and then after this completes, it updates a remote flag. The remote processor spins until this flag is updated before proceeding. This is in contrast to Fortran, which depends on the synchronization between sender and receiver to notify the remote processor as to when the data transfer is finished. Therefore, communication is a factor in Titanium's slower overall CG performance.

5.2.5 Serial Performance

Now we examine whether computation is also a factor in this performance gap. As seen in figure 21, this is indeed the case. Titanium's serial running times are greater than Fortran's times over all platforms. The majority of the computation is sparse matrix-vector products, which entails indirect memory accesses. The Fortran code is generating faster code for performing these indirect accesses than the Titanium code. In general, we have seen that serial Fortran running times on the Itanium and Alpha are significantly faster than

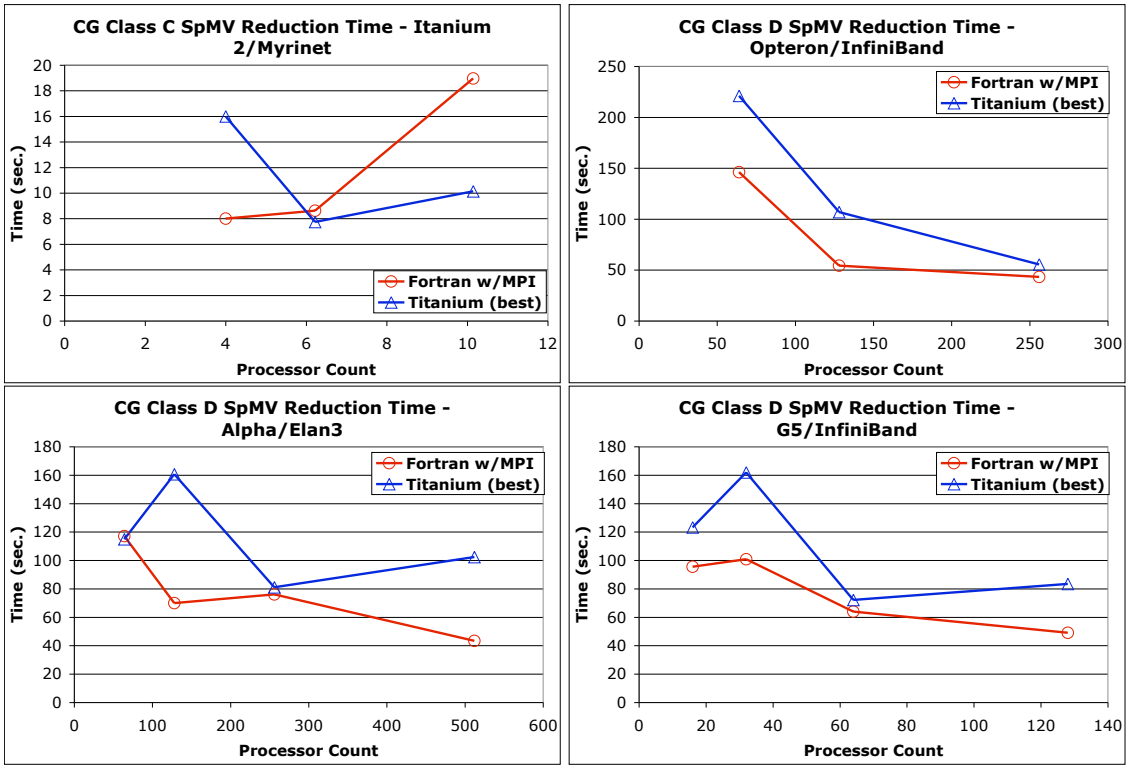


Figure 20: A comparison of the mean time (over all processors) spent in SpMV reduction by Titanium and Fortran with MPI.

Titanium, but on the Opteron and the G5 platforms, the running times are similar. These results are another confirmation of that.

6 Related Work

The prior work on parallel languages is too extensive to survey here, so we focus on three current language efforts (ZPL, CAF, and UPC) for which similar studies of the NAS Parallel Benchmarks have been published. All of these studies consider performance as well as expressiveness of the languages, often based on the far-from-perfect line count analysis that appears here.

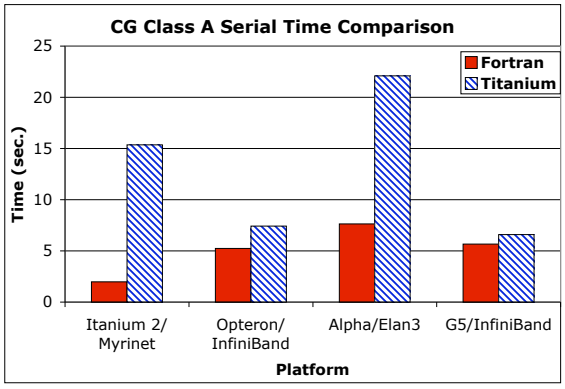


Figure 21: A comparison of serial running times between Titanium and Fortran. This is a good measure of computational speed.

6.1 ZPL

ZPL is a data parallel language developed at the University of Washington. A case study by Chamberlain, Deitz and Snyder [6, 7] compared implementations of NAS MG across various machines and parallel languages (including Fortran/MPI, ZPL [26], Co-Array Fortran [22], High Performance Fortran [14], and Single-Assignment C). They compared the implementations in terms of running time, code complexity and conciseness. Our work provides a similar evaluation of Titanium for MG, but then extends their work by including two other NAS benchmarks.

One of the major conclusions of their study was a suggested correlation between language expressiveness and a language feature they refer to as a "global view" of computation. Specifically, they find that "languages with a local view of computation require 2 to 8 times as many lines of code as those providing a global view, and that the majority of these lines implement communication." By their definition, Titanium provides a local view, because loops execute over a local portion of a data structure, rather than globally. In contrast, a data parallel language like ZPL expresses computation at a global level and requires the compiler to convert this into local computation.

Our work demonstrates that "local view" languages (like Titanium) need not suffer in conciseness, even when compared to benchmarks that are naturally expressed in the more restrictive data-parallel model. For instance, the MG code size is comparable between the two languages (192 lines total in ZPL, compared to 197 lines in Titanium). The construction of distributed data structures does generally require slightly more application code in Titanium than ZPL, because the partitioning of global indices into local ones is explicit. However, the inclusion of Titanium's powerful domain abstraction makes this code sim-

ple and concise. Moreover, the generality of Titanium's distribution mechanism means that it can easily express irregular, distributed, pointer-based data structures that would be cumbersome at best in data-parallel languages.

At the time the ZPL study was done, there was little evidence regarding the performance of one-sided communication, and their expectation was that the one-sided communication model would not perform well on distributed memory architectures. Our work on GASNet [15] and that of Nieplocha et al on ARMCI [21] show that in fact one-sided communication can often outperform two-sided message-passing communication. Moreover, the results of this paper in the context of Titanium and others in the context of CAF [23] and UPC [2] show that these performance advantages carry over to application-level performance in the NAS benchmarks.

6.2 Co-Array Fortran

Co-Array Fortran (CAF) is an explicitly parallel, SPMD global address space extension to Fortran 90 initially developed at Cray Inc [22]. A compiler is available for the Cray X1, and an open-source compiler for a dialect of CAF is available from Rice University. The Rice compiler translates CAF programs to Fortran 90 with calls to a communication system based on GASNet or ARMCI [20]. The Rice CAF compiler has been used in several studies with the NAS Parallel Benchmarks, demonstrating performance comparable to, and often better than, the Fortran/MPI implementations [8, 10, 23].

CAF has a built-in distributed data structure abstraction that specifies the distribution by identifying a co-dimension that is spread over the processors. Therefore, layouts are more restrictive than in languages like ZPL, HPF, and certainly Titanium. However, communication is more visible in CAF than most languages because only statements involving the co-dimension can result in communication. Because CAF is based on F90 arrays, it has various array statements (which are not supported in Titanium) and subarray operations (which are). Although the ZPL study includes a CAF implementation of MG, the implementation used was heavily influenced by the original Fortran/MPI MG code, and therefore expectedly had comparable length. In contrast, our Titanium implementations were written from scratch to best utilize the available language features and demonstrate the productivity advantages.

6.3 UPC

Unified Parallel C (UPC) [27] is a parallel extension of ISO C99 [5] that provides a global memory abstraction and communication paradigm similar to Titanium. UPC currently enjoys the most widespread support of the PGAS languages, with a number of vendor

implementations and two open-source implementations. The Berkeley UPC [3, 9] and Intrepid UPC [17] compilers use the same GASNet communication layer as Titanium, and Berkeley UPC uses a source-to-source compilation strategy analogous to the Berkeley Titanium compiler and Rice CAF compiler.

El Ghazawi et al. [12] ported the Fortran/MPI versions of the NAS Parallel Benchmarks into UPC, with mixed performance results relative to MPI. Bell et al [2] reimplemented some of the NAS parallel benchmarks from scratch in UPC, using one-sided communication paradigms and optimizations made possible by the Partitioned Global Address Space abstraction. These implementations delivered performance improvements of up to 2x over the Fortran/MPI implementations.

7 Conclusions

As we have shown, Titanium's features are well-suited to express the common scientific paradigms found in the NAS Parallel Benchmarks: nearest neighbor computation on a 3D mesh (MG), FFT with an all-to-all transpose on a 3D mesh (FT), and 2D sparse matrices with indirect array accesses (CG). As a rough indication of expressiveness, figure 2 shows that Titanium code is significantly shorter than Fortran w/MPI. However, Titanium's features are not limited to these problems, as they actually support more general distributed data layouts and irregular parallelism patterns than these problems require.

Concerning performance, computation and communication need to be examined separately. As seen in the serial performance comparisons with Fortran, Titanium's computational speed is comparable on the G5 and Opteron platforms, but significantly slower on the Alpha and Itanium machines across all three benchmarks. This is at least partly attributable to the quality of the vendor-supplied C compiler used to compile Titanium's generated C code.

As for communication, Titanium's one-sided message passing model allows for a modification in the FT algorithm so that much of the all-to-all communication is overlapped with other computation. This results in a large performance gain over the Fortran code. In the MG benchmark, there is not as much opportunity for such overlap, but overall, the communication time slightly favors Titanium. However, Titanium's one-sided communication becomes a disadvantage in the CG benchmark. During the CG reduction step, a remote flag has to be set after each data transfer so that the remote processor knows to proceed. This extra communication prevents Titanium from doing as well as MPI. However, there are solutions for this within the one-sided communication model, and is a project for future work.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[3] The Berkeley UPC Compiler, 2002. http://upc.lbl.gov.

[4] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment, September 1988.

[5] Programming Languages – C, 1999. The ISO C Standard, ISO/IEC 9899:1999.

[6] B. L. Chamberlain, S. Deitz, and L. Snyder. Parallel language support for multi-grid algorithms. Technical Report UW-CSE 99-11-03, University of Washington, November 1999.

[7] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[8] D. Chavarria-Miranda, C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanty, and Y. Yao. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[9] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.

[10] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[11] K. Datta, D. Bonachea, and K. Yelick. Titanium Performance and Potential: an NPB Experimental Study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2005.

[12] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Supercomputing2002 (SC2002)*, November 2002.

[13] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[14] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS parallel benchmarks in high performance fortran. Technical Report NAS-98-009, Nasa Ames Research Center, Moffet Field, CA, September 1998.

[15] GASNet home page. http://gasnet.cs.berkeley.edu/.

[16] E. Givelberg and K. Yelick. Distributed Immersed Boundary Simulation in Titanium, 2003.

[17] Intrepid Technology, Inc. *GCC/UPC Compiler*. http://www.intrepid.com/upc/.

[18] S. Merchant. Analysis of a Contractile Torus Simulation in Titanium, August 2003.

[19] MPI Forum. MPI: A message-passing interface standard, v1.1. Technical report, University of Tennessee, Knoxville, June 12, 1995. http://www.mpi-forum.org/docs/mpi-11.ps.

[20] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proc. RTSPP IPPS/SDP'99*, 1999.

[21] J. Nieplocha, V. Tipparaju, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02, Ft Lauderdale, FL*, 2002.

[22] R. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[23] R. W. Numrich, J. Reid, and K. Kim. Writing a multigrid solver using co-array fortran. In *Proceedings of the Fourth International Workshop on Applied Parallel Computing, Umea, Sweden*, June 1998.

[24] G. Pike and P. N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of the IEEE/ACM SC2002 Conference*, 2002.

[25] G. R. Pike. Reordering and Storage Optimizations for Scientific Programs, January 2002.

[26] L. Snyder. *The ZPL Programmer's Guide*. MIT Press, 1999.

[27] UPC Community Forum. *UPC specification v1.2*, 2005. http://upc.gwu.edu/documentation.html.

[28] T. Wen and P. Colella. Adaptive Mesh Refinement in Titanium. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

A Experimental Platforms

	Itanium 2/ Myrinet	Opteron/ InfiniBand	Alpha/ Elan3	G5/ InfiniBand
Location	UC Berkeley/ CITRIS Cluster	NERSC/ Jacquard	PSC/ Lemieux	Virginia Tech/ System X
Node Count	32	320	750	1100
Node Type	Dual 1.3 GHz Itanium 2	Dual 2.2 GHz Opteron	Quad 1 GHz Alpha 21264C	Dual 2.3 GHz PowerPC G5
Proc FPUs	2	2	2 (one for mult)	2
Fused Mult-Add	yes	no	no	yes
Proc Peak	5.2 GFlops/s	4.4 GFlops/s	2.0 GFlops/s	9.2 GFlops/s
L1 Data Cache	32 KB (no float)	64 KB	64 KB	32 KB
L2 Cache	256 KB	1 MB	8 MB (off-chip)	512 KB
L3 Cache	3 MB (on-chip)	None	None	None
Node Memory	4 GB	4 GB	4 GB	4 GB
Titanium compiler	v3.86	v3.74	v3.74	v3.74
Titanium back-end C compiler	Intel C v9.0	Pathscale v2.0	Compaq C v6.5	IBM XLC v6.0 for OS X
Fortran compiler	Intel Fortran v9.0	Intel Fortran v8.1	HP Fortran v5.5A-3548	IBM XLF v8.1
OS Software	Linux 2.4.20	Linux Enterprise Server 9	Compaq Tru64 UNIX	Apple Mac OS X 10.3.9
Network	Myricom Myrinet LANai XP PCI-X M3F-PCIXD-2	Mellanox Cougar InfiniBand 4x HCA	Quadrics QsNet1 Elan3 w/ dual rail (one rail used)	Mellanox Cougar InfiniBand 4x HCA

B Problem Classes

Benchmark	Class	Matrix/Grid Dim	Iterations	Parallel Partitioning
CG	A	$(1.4 \cdot 10^4)^2$	15	2D partitioning– Equal blocks of sparse matrix
	B	$(7.5 \cdot 10^4)^2$	75	
	C	$(1.5 \cdot 10^5)^2$	75	
	D	$(1.5 \cdot 10^6)^2$	100	
FT	A	$256^2 \cdot 128$	6	1D partitioning– Equal slabs of grid
	B	$512 \cdot 256^2$	20	
	C	512^3	20	
	D	$2048 \cdot 1024^2$	25	
MG	A	256^3	4	3D partitioning– Equal blocks of grid
	B	256^3	20	
	C	512^3	20	
	D	1024^3	50	