Titanium Performance and Potential: an NPB Experimental Study

Kaushik Datta[1], Dan Bonachea[1] and Katherine Yelick[1,2]
{*kdatta,bonachea,yelick*}*@cs.berkeley.edu*
Computer Science Division, University of California at Berkeley[1]
Lawrence Berkeley National Laboratory[2]

Abstract

Titanium is an explicitly parallel dialect of Java^{TM} 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 standard benchmark suite of kernels that are common across many scientific applications. We argue that parallel languages like Titanium provide greater expressive power than conventional approaches, enabling much more concise and expressive code and minimizing time to solution without sacrificing parallel performance. Empirical results demonstrate our Titanium implementations of three of the NAS Parallel Benchmarks can match or even exceed the performance of the standard MPI/Fortran implementations at realistic problem sizes and processor scales, while still using far cleaner, shorter and more maintainable code.

1 Introduction

The tension between programmability and performance in software development is nowhere as acute as in the domain of high end parallel computing. The entire motivation for parallelism is high performance, so programmers are reluctant to use languages that give control to compilers or runtime systems. Yet the difficulty of programming large-scale parallel machines is notorious—it limits their marketability, hinders exploration of advanced algorithms, and restricts the set of available programmers.

The Titanium language was designed to address these issues, providing programmers with high level program structuring techniques, yet giving them control over key features of parallel performance: data layout, load balancing, identification of parallelism, and synchronization. In this paper we describe our experience using Titanium on a standard set of application benchmarks taken from the NAS Parallel Benchmark suite [3]. These benchmarks are representative of programming paradigms common to many scientific applications, although they do not reflect features of some full applications, such as adaptivity, multiple physical models, or dynamic load balancing. Titanium has been demonstrated on these more complete and more general application problems [21, 46]. In this paper we study three of the NAS benchmarks, revealing some of the key programming and performance features of Titanium and allowing direct comparisons with codes written in the more common message passing model.

Modern parallel architectures can be roughly divided into two categories based on the programming interface exposed by the hardware: shared memory systems where parallel threads of control all share a single logical memory space (and communication is achieved through simple loads and stores), and distributed memory systems where some (but not necessarily all) threads of control have disjoint memory spaces and communicate through explicit communication operations (e.g. message passing). Experience has shown that the shared memory model is often easier to program, but it presents serious scalability challenges to hardware designers. Thus, with a few notable exceptions, distributed memory machines currently dominate the high-end supercomputing market.

The Partitioned Global Address Space (PGAS) model seeks to combine the advantages of both shared and distributed memory. It offers the programmability advantages of a globally shared address space, but is carefully designed to allow efficient implementation on distributed-memory architectures. Titanium [49], UPC [45] and Co-array Fortran [36] are examples of modern programming languages that provide a global address space memory model, along with an explicitly parallel SPMD control model. PGAS languages typically make the distinction between local and remote memory references explicitly visible to encourage programmers to consider the locality properties of their program, which can have a noticeable performance impact on distributed memory hardware.

One major contribution of the paper is to showcase the performance and productivity benefits of the Titanium programming language. We demonstrate by example that scientific programming in PGAS languages such as Titanium can provide major productivity improvements over programming with serial languages augmented with a message-passing library. Furthermore, we show evidence that programming models with one-sided communication (such as that used in PGAS languages) can achieve application performance comparable to or better than similar codes written using two-sided message passing, even on distributed memory platforms.

The rest of the paper is organized as follows. In sections 2 and 3, we provide background information on the Titanium language and introduce the NAS Parallel Benchmarks. Sections 4, 5 and 6 describe our use of Titanium language features for implementing the NAS benchmarks. Section 7 presents empirical performance results for the Titanium implementations of CG, MG and FT. Section 8 provides an overview of related work, and we conclude in section 9.

2 Titanium Overview

Titanium [49] is an explicitly parallel, SPMD dialect of Java^{TM} that provides a Partitioned Global Address Space (PGAS) memory model, as shown in figure 1. Titanium supports the creation of complicated data structures and abstractions using the object-oriented class mechanism of Java, augmented with a global address space to allow for the creation of large, distributed shared structures. As Titanium is essentially a superset of Java [22], it inherits all the expressiveness, usability and safety properties of that language. In addition, Titanium notably adds a number of features to standard Java that are designed to support high-performance computing. These features are described in detail in the Titanium language reference [23] and include:

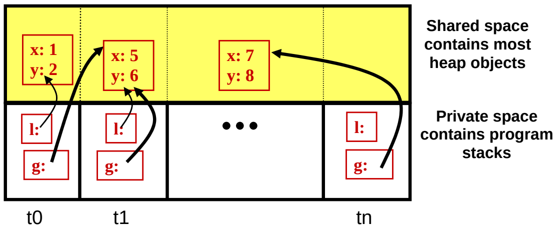
x: 1
y: 2
x: 5
y: 6
x: 7
y: 8
l:
g:
l:
g:
•••
l:
g:
t0
t1
tn
Shared space contains most heap objects
Private space contains program stacks

Figure 1. Titanium's PGAS Memory Model.

1. **Flexible and efficient multi-dimensional arrays** - Java's language support for arrays is limited to single-dimensional arrays (and 1-D arrays of 1-D arrays), an issue which negatively impacts the runtime performance and optimizability of Java programs. Titanium's multidimensional array extensions add support for high-performance hierarchical and adaptive grid-based computations, providing a flexible and expressive set of slicing, restriction and transposition operators, which are amenable to aggressive compiler optimization. Most of the bulk communication in well-tuned Titanium applications is expressed as rectangular copy operations over these multi-dimensional arrays (e.g., a ghost value exchange).
2. **Built-in support for multi-dimensional domain calculus** - Titanium includes language support for performing concise domain calculus, providing N-dimensional points, rectangles and arbitrary domains as first-class types and literals. These types are used when indexing and manipulating the multi-dimensional arrays, providing powerful expressiveness for regular and adaptive grid-based applications.
3. **Locality and sharing reference qualifiers** - Titanium's type system is augmented with reference qualifiers that enable the expression and inference of locality and sharing properties of distributed data structures [29, 30].
4. **Explicitly unordered loop iteration** - Titanium's *foreach* loop construct provides a concise means for expressing iteration over an N-dimensional region or array domain and includes an explicitly unordered iteration semantic that enables aggressive optimization on vectorizable loops, as typified by many stencil computations.
5. **User-defined immutable classes** - The Titanium immutable class extension provides language support for user-defined primitive types (often called "lightweight" or "value" classes), which enables the creation of user-defined unboxed objects, analogous to C structs. These provide efficient support for extending the language with new types which are manipulated and passed by value (e.g., for implementing complex numbers), avoiding pointer-chasing overheads often associated with tiny objects, without sacrificing the productivity advantages of data abstraction and object-oriented programming.
6. **Operator-overloading** - Titanium includes a convenience feature that allows programmers to define custom operators over user-defined classes, allowing improved conciseness and readability.
7. **Cross-language support** - Titanium provides support for cross-language applications, enabling code reuse through the use of external computational kernels or libraries.

Titanium adds several other major features relative to Java which will not be further addressed in this paper due to length limitations. These include:

1. **Templates** - Titanium proves support for parameterized classes, including support for primitive types as template actuals (a feature which is lacking in recent template extensions to Java, and which we find to be important for performance of scientific computing).
2. **Zone-based memory management** - Titanium complements the standard Java garbage collector and automatic memory management facilities with safe support for explicit memory management through the use of reference-counted memory regions [2].
3. **Compile-time prevention of deadlocks on barrier synchronization** - Titanium augments Java's safety features by providing checked synchronization that conservatively prevents certain classes of collective synchronization bugs at compile-time [1].
4. **Library support for large-scale parallelism** - In addition to the standard Java libraries, Titanium also provides several new libraries that expose useful parallel synchronization primitives and collective operations.

Major applications have been written and continue to be developed in Titanium, including immersed boundary simulations of the heart [32, 47, 48] and cochlea [21], adaptive mesh refinement (AMR) solvers for both hyperbolic and elliptic problems [4, 31, 39], and a Titanium implementation of the CHOMBO AMR library [46].

Since Titanium is an explicitly parallel language, new analyses are often needed even for standard code motion transformations. Titanium compiler research focuses on the design of program analysis techniques for parallel code [1, 25, 26, 27, 28], and on optimizing transformations for distributed memory environments [42, 43]. There are also analyses and optimizations for single processor loop tiling [38, 40] and automatically identifying references to objects on the local processor [29, 30].

The current Titanium compiler implementation [44] uses a static compilation strategy - programs are translated to intermediate C code and then compiled to machine code using a vendor-provided C compiler and linked to native runtime libraries which implement communication, garbage collection, and other system-level activities. There is no JVM, no JIT, and no dynamic class loading. Titanium's global memory space abstraction allows parallel processes to directly reference each other's memory to read and write values or arrange for bulk data transfers, and all communication is one-sided (i.e., there is no explicit involvement by the remote process and no need for a matching "receive" operation). Titanium is extremely portable, and Titanium programs can run unmodified on uniprocessors, shared memory machines and distributed memory machines. Incremental performance tuning may be necessary to arrange an application's data structures for efficient execution on distributed memory hardware, but the functional portability allows for development on shared memory machines and uniprocessors.

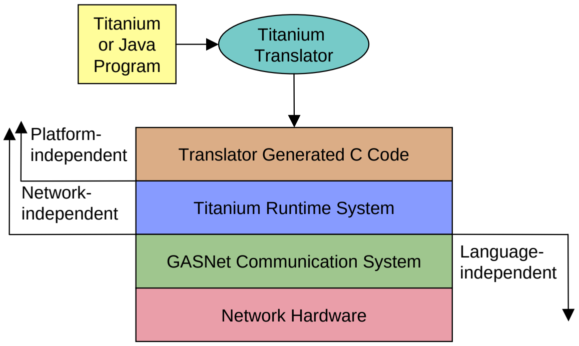

Figure 2. High-level System Architecture of the Titanium Implementation

The current implementation runs on a large range of platforms, including uniprocessors, shared memory multiprocessors, distributed-memory clusters of uniprocessors or SMPs, and a number of specific supercomputer architectures (Cray X1/T3E, IBM SP, SGI Altix/Origin). Figure 2 illustrates the high-level organization of the implementation on distributed memory backends, where communication is implemented using calls to the GASNet [9] communication system. GASNet provides portable, high-performance, one-sided communication operations tailored for implementing PGAS languages such as Titanium across a wide variety of network interconnects. GASNet has been implemented natively on Myrinet (GM), Quadrics QsNetI/QsNetII (elan3/4), InfiniBand (Mellanox VAPI), IBM SP Colony/Federation (LAPI), Dolphin (SISCI), Cray X1 (shmem) [6] and SGI Altix (shmem). There are also implementations for MPI 1.1 and UDP (e.g., Ethernet), which are not intended for high performance runs, but provide near-universal portability.

3 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks consist of a set of kernel computations and larger pseudo applications, taken primarily from the field of computational fluid dynamics [3]. In this paper we use three of the kernel benchmarks: MG, CG, and FT. MG is a multigrid computation involving hierarchical regular 3-D meshes partitioned in a 3-D blocked layout. CG is a sparse iterative solver, dominated by a sparse matrix-vector multiplication on a randomly generated sparse matrix distributed in a 2-D blocked layout. FT performs a series of 3-D FFTs on a 3-D mesh that is partitioned across processors in one dimension, using local 1-D FFTs in each dimension and a global grid transpose. These three benchmarks reflect three different kinds of computation and communication patterns that are important across a wide range of applications.

The original reference implementation of the NAS Parallel Benchmarks is written in serial Fortran and MPI [33]. We use these implementations as the baseline for comparison in this study. MPI represents both the predominant paradigm for large-scale parallel programming and the target of much concern over productivity, since it often requires tedious packing of user level data structures into aggregated messages to achieve acceptable performance. In addition, the MPI version 1 variant used in the majority of deployed MPI codes (including the reference implementation of the NAS Parallel Benchmarks) requires the programmer to coordinate all communication using two-sided message-passing operations, which tend to obfuscate and complicate the communication pattern, reducing productivity and stifling innovation. Global Address Space languages such as Titanium are an attempt to find a suitable alternative for MPI that addresses these shortcomings, while still providing comparable or better performance on applications of interest.

4 Titanium Features in the Multigrid (MG) 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.

4.1 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 1-D arrays to represent the vectors and a set of 1-D arrays to represent a sparse matrix; both MG and FT use 3-D 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 3-D 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 non-ghost 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, 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);
     }
```

A common class of loop bounds and indexing errors are avoided by having the compiler and runtime system keep track of the iteration boundaries for the multidimensional traversal. The `applyStencil` method may safely refer to elements that are 1 point away from `p`, since the loop is over the interior of a larger array, and this one loop concisely expresses iteration over multiple dimensions, corresponding to a multi-level loop nest in other languages.

4.2 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 2-D 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 2-D 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 = [1,0], SOUTH = [-1,0], EAST = [0,1], WEST = [0,-1];

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

The full MG code used for benchmarking in section 7 includes a 27-point stencil applied to 3-D arrays, and the Titanium code, like the Fortran code, uses a manually-applied stencil optimization that eliminates redundant common subexpressions, a key optimization for the MG benchmark [13].

4.3 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 [13]. The generality of Titanium's distributed data structures are not fully utilized in the NAS benchmarks, because the data structures are simple distributed arrays, rather than trees, graphs or adaptive structures [46]. 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.

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 3-D 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 a local block. Every processor then allocates the 1-D array `blocks`, in which each element is a reference to a 3-D array. The local blocks are then combined into a distributed data structure using the `exchange` operation, which performs a gather-to-all communication that stores each processor's contribution in the corresponding element of the `blocks` array. Figure 3 illustrates the resulting data structure for a 3-processor execution.

Now `blocks` is a distributed data structure, but it maps a 1-D array of processors to blocks of a 3-D grid. To create a more natural mapping, a 3-D array called `blocks3D` is introduced. It uses `blocks` and a method called `procForBlockPosition` (not shown) to establish an intuitive mapping from a 3-D array of processor coordinates to blocks in a 3-D grid. Note that the indices for each block use global coordinates, and that these blocks are designed to overlap on some indices; these overlapped areas will serve as ghost regions.

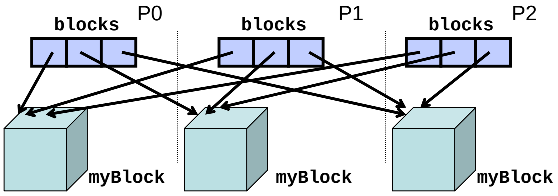

Figure 3. Distributed data structure built for MG.

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 can be used to express a set of discontiguous blocks, as in AMR codes, or an arbitrary set of objects, and is not restricted to arrays. 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.

4.4 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. The ghost cells, a set of array elements surrounding the actual grid, cache values belonging to the neighboring grids in the physical space of the simulation. Since adjacent grids have consecutive non-ghost cells, each grid's ghost cells will overlap the neighboring grid's non-ghost 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:

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

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. The use of the global indexing space in the grids of the distributed data structure (made possible by the arbitrary index bounds feature of Titanium arrays) makes it easy to select and copy the cells in the ghost region, and is also used 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. The MG benchmarks uses periodic boundary conditions, and an additional array view operation is required before the copy to logically translate the array elements to their corresponding elements across the domain:

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

The translate method translates the indices of the array view, creating a new view where the relevant points overlap their corresponding non-ghost cells in the subsequent copy.

4.5 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 which 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, potentially eliminating some unnecessary overheads in array access (for example, dynamic checks of whether a given global array access references data that actually resides locally and thus requires no communication). 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.

4.6 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 3-D 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 the partitioned global address space memory abstraction relieves programmers from the traditionally error-prone tedium of expressing the communication of the MG algorithm using two-sided message passing in MPI.

Figure 4 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 with 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.

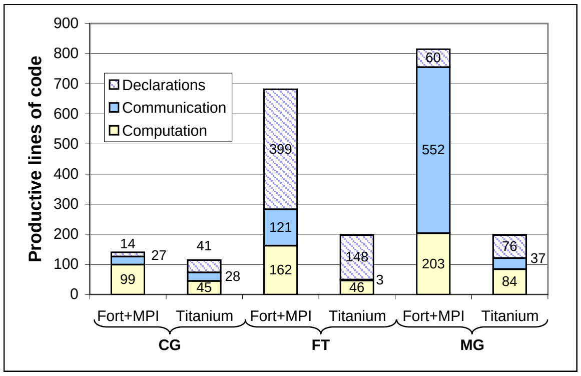

Figure 4. Timed region line count comparison

While the Titanium MG code is algorithmically similar to the NAS MG code, it is completely rewritten in the Titanium paradigm. The only real algorithmic difference is that the NAS 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 and reduce overall running time, but it slightly increases the Titanium line count. Titanium's expressive power makes it significantly easier to experiment with such communication-related algorithmic optimizations, reducing tuning time and increasing productivity.

5 Titanium Features in the Conjugate Gradient (CG) Benchmark

The following sections highlight the Titanium features used to implement the Conjugate Gradient (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 Foreach Loops

As described in section 4.2, 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 automatic cache blocking and tiling optimizations [38, 40]. 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 (Compressed Sparse Row) format, with data structures illustrated in Figure 5. The `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 five non-zero data values. The data values are stored contiguously in the array *a*.

Arrays *rowRectDomains* and *colIdx* hold the metadata used to perform sparse indexing. Arrows indicate the correspondence of coordinates for the non-zero value which logically resides at row 2, column 12 of the sparse array.

```
// 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;
} }
```

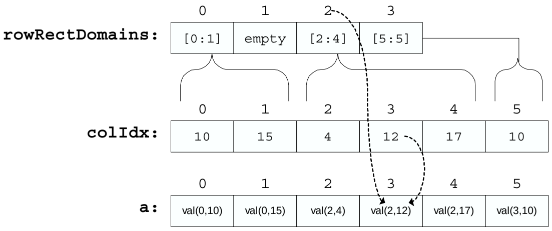

Figure 5. CSR format for sparse array data.

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 may affect the exact result, it does not affect algorithm correctness for reasonable matrices.

5.2 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 on a subset of processors (a *team reduction*) to perform a dot product. The code performs this reduction using a pairwise exchange algorithm with $\log n$ phases, and synchronization between processor pairs is required as data is passed in each phase of the reduction algorithm. 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 a faster method is to employ more direct and efficient techniques for point-to-point synchronization between pairs of processors during each phase. 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 operation is still one-sided in flavor – specifically, the initiator (the data producer) 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. Consequently, 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 – consequently 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 non-blocking communication can be used in some algorithms to overlap most of this latency with independent computation and other data transfers, however in the specific case of CG's pairwise-exchange reduction 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

Java Version

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

Titanium Version

```
public immutable class Complex { // <-- note 'immutable'
  public double real, imag;
  public inline Complex(double r, double i)
    { real = r; imag = i; }
  public inline Complex op+(Complex c)
    { return new Complex(c.real + real, c.imag + imag);  }
  public inline Complex op*(double d)
    { return new Complex(c.real * d, c.imag * d);  }
  ...
}


   /* sample usage */
Complex c = new Complex(7.1, 4.3);
Complex c2 = (c + c) * 14.7;
```

Figure 6. Complex numbers in Java and Titanium

library so it could be used in the parallel dot product on a subset of processors.

5.3 The CG Benchmark Implementation

The CG (Conjugate Gradient) benchmark demonstrates the use of Titanium arrays and foreach loops for implementing the data structures and computations on a sparse matrix stored in CSR format, as described in the previous sections. The team reduction operations required by the 2-D blocked decomposition motivate the use of point-to-point synchronization constructs to achieve efficient pairwise synchronization in producer-consumer communication patterns, a relatively new area of exploration in the context of PGAS languages such as Titanium where all communication is fully one-sided and implicit.

Figure 4 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. Also, MPI's message passing semantics implicitly provide pairwise synchronization between message producers and consumers, so no additional code is required to achieve that synchronization. It is worth noting that message-passing applications are forced to pay the overheads associated with message matching synchronization, whether that synchronization is required by the application semantics or not. As shown in our previous work [5], this factor measurably reduces the achievable communication bandwidth relative to one-sided communication models, most notably at medium-sized messages where there is insufficient user data in each message to amortize the per-message MPI overheads associated with message matching and buffer management.

The Titanium implementation is comparably concise – despite the fact that extra code is required to achieve the pairwise synchronization, the amount of communication code is roughly equivalent. The computation code is 54 lines shorter in Titanium and there are 27 additional declaration lines – this difference is due to the use of object-oriented constructs and modularity to implement the vector manipulation operations as methods in a Titanium Vector class (leading to improved conciseness and readability of the computation code that orchestrates the vector computations), whereas those same operations are implemented in the Fortran code by expanding the vector operations inline into the caller as many very similar do loops, leading to expansion and obfuscation of the computation code.

6 Titanium Features in the Fourier Transform (FT) Benchmark

The FT benchmark illustrates several useful Titanium features, both in terms of 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*. Finally, to further help performance, Titanium supports both *cross-language calls* and *nonblocking array copy*.

6.1 Immutables and Operator Overloading

The Titanium immutable class feature provides language support for defining application-specific primitive types (often called "lightweight" or "value" classes) - allowing the creation of user-defined unboxed objects, analogous to C structs. These provide efficient support for extending the language with new types which are manipulated and passed by value, avoiding pointer-chasing overheads which 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 6 compares how one might define a Complex number class using either standard Java Objects versus 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 –

imposing the overhead of storing a pointer and an Object header for each complex number, which can easily double the required storage space for each such entity. More importantly for the purposes of scientific computing, such a representation induces poor memory locality and cache behavior for operations over large arrays of such objects. Finally, note the cumbersome method-call syntax which is required for performing operations on the Complex Objects in standard Java.

Titanium allows easy resolution of these 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 passed by value and 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. Also they are 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. Finally, note the optional use of Titanium's *inline* method modifier, a hint to the optimizer that calls to the given method should be inlined into the caller (analogous to the C++ inline modifier).

6.2 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 [18] library to perform the local 1-D FFT computations, thereby leveraging the auto-tuning features and machine-specific optimizations made available in that off-the-shelf FFT kernel implementation. Note that although the FFTW library does offer a 3-D MPI-based parallel FFT solver, our benchmark only uses the serial 1-D 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 3-D Titanium arrays from within native methods, where their representation as 1-D 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 3-D array data is stored natively in a row-major, contiguous layout. Java's layout of "multidimensional" arrays as 1-D arrays of pointers to 1-D arrays implies discontiguity of the array data that would significantly have increased the computational costs and complexity associated with calling external multidimensional computational kernels like FFTW.

6.3 Nonblocking Arraycopy

Titanium's explicitly nonblocking array copy library methods helped in implementing a more efficient 3-D FFT. Both the Fortran+MPI and the Titanium 3-D FFT algorithms are illustrated in figure 7.

The Fortran code performs a bulk-synchronous 3-D FFT, whereby each processor performs two local 1-D FFTs, then all the processors collectively perform an all-to-all communication, followed by another local 1-D 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 dealt with using a slight reorganization of the 3-D FFT algorithm employing nonblocking array copy. The new algorithm, implemented in Titanium, first performs a local strided 1-D FFT, followed by a local non-strided

[1] The Berkeley Titanium compiler uses the Boehm-Weiser conservative garbage collector [8] 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 which are typically used by standard Java implementations.

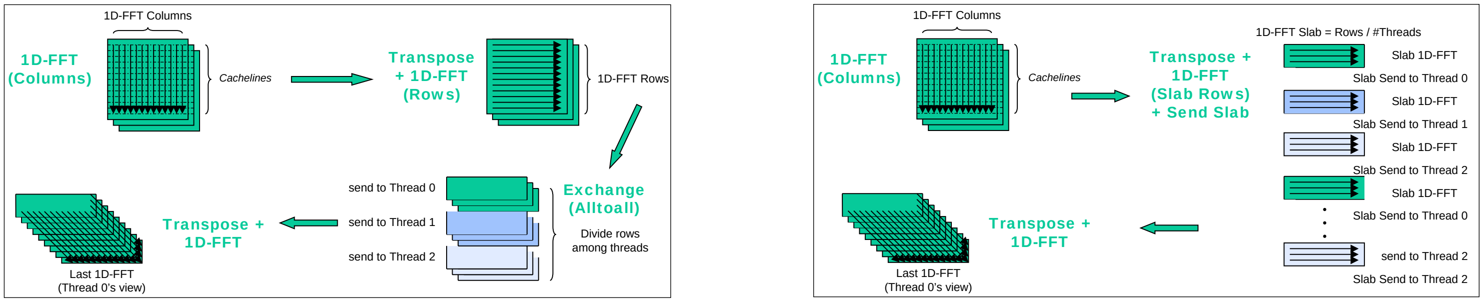

Figure 7. Illustration of the blocking exchange algorithm for communication in FT, used by the stock Fortran+MPI code, and the non-blocking overlapped slabs algorithm for communication in FT, which provided the best performance in the Titanium code.

1-D FFT. Then, we begin sending each processor's portion of the grid (slab) as soon as the corresponding rows are computed. By staggering the messages throughout the computation, the network is less likely to become congested and is more effectively 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 8.

6.4 The FT Benchmark Implementation

The previous sections demonstrate how the FT benchmark exploits several powerful Titanium features. First, the use of immutables in the Complex number class eliminated extra levels of indirection and additional memory overhead, resulting in major performance benefits. The Complex number class was then simplified using Titanium's operator overloading feature – this made arithmetic operations on Complex numbers cleaner and more intuitive.

Two other features helped further optimize performance. First, Titanium's ability to make cross-language calls, coupled with its support for true multidimensional arrays, allowed us to take advantage of the highly-tuned FFTW library. Secondly, nonblocking array copy allowed us to implement a more efficient 3-D FFT algorithm that both reduced network congestion and overlapped communication with computation.

In terms of code size, figure 4 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 1-D 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 1-D 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 for Fortran, resulting in a 121 to 3 disparity in lines of communication.

7 Performance Results

7.1 Experimental Methodology

In order to compare performance between languages, we measured the performance of the Titanium and Fortran with MPI implementations on a G5 cluster and an Opteron cluster, both with an InfiniBand interconnect. Complete platform details are shown in the Appendix. See the NAS [3] benchmark specification for details concerning input sizes for each problem class.

During data collection, each data point was run consecutively three times, with the minimum 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.

Note that all speedups are measured against the base case of the best time at the lowest number of processors for that graph, and the absolute performance of that case is shown on the y axis. Therefore, if the Titanium code performed better than the Fortran code at the lowest number of processors, then *both* the Titanium and Fortran speedups are computed relative to that same timing.

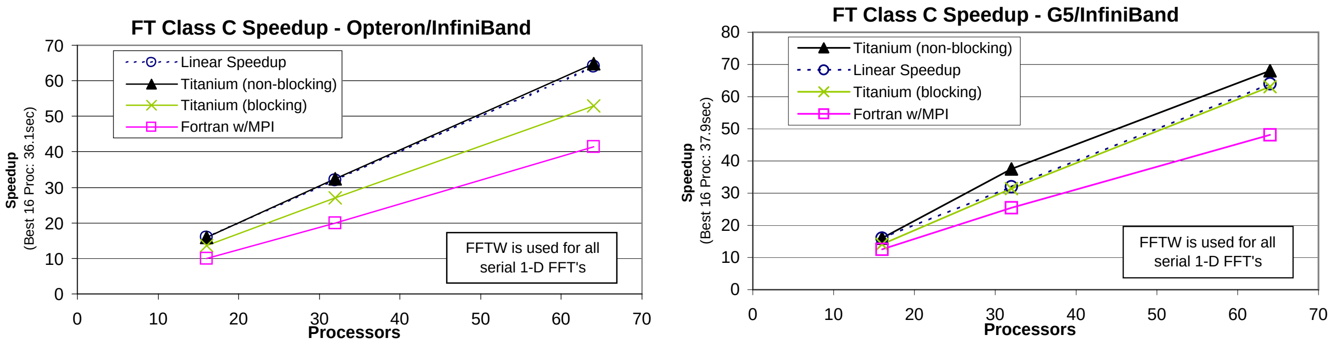

Figure 8. FT performance comparison

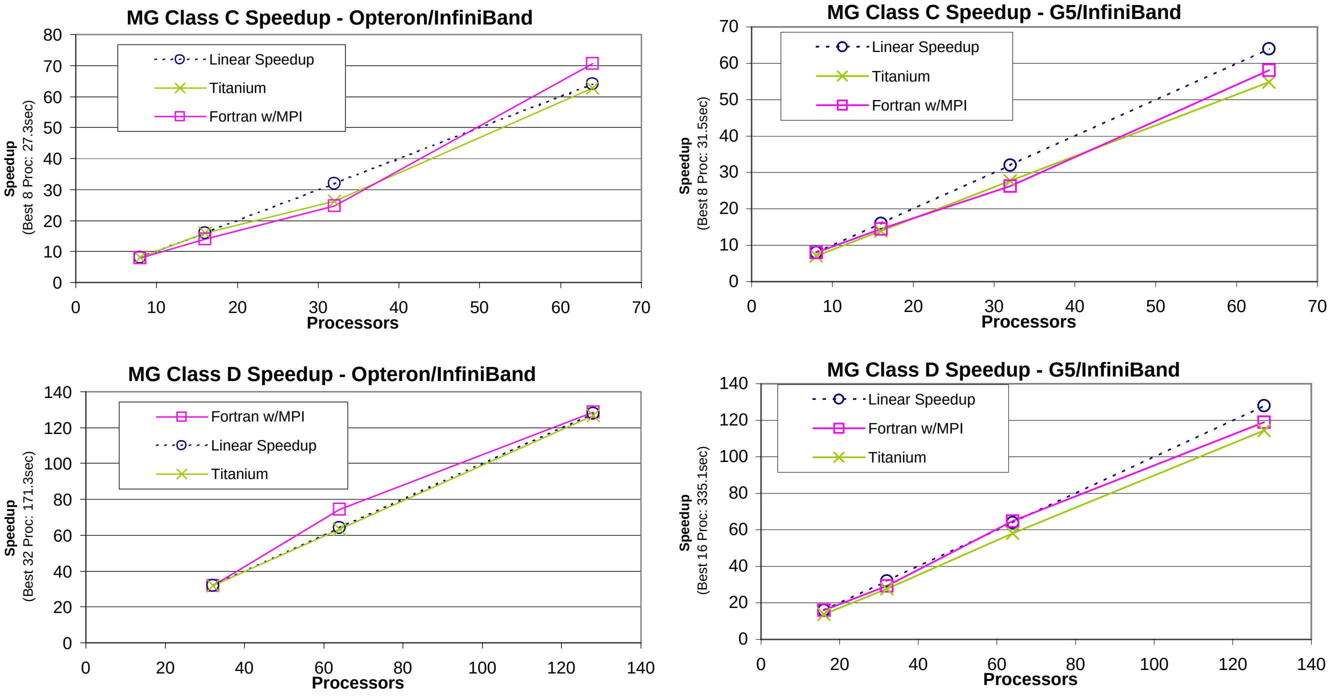

Figure 9. MG performance comparison

7.2 FT Performance

Both implementations of the FT benchmark use the same version of the FFTW library [18] for the local 1-D FFT computations, since it was found to always outperform the local FFT implementation in the stock Fortran implementation. However, all the communication and other supporting code is written in the language being examined.

As seen in figure 8, the Titanium FT benchmark thoroughly outperforms Fortran, primarily due to two optimizations. First, the Titanium code uses padded arrays to avoid the cache-thrashing that results from having a power-of-two number of elements in the contiguous array dimension. This helps to explain the performance gap between Fortran and the blocking Titanium code.

Secondly, as explained in section 6 the best Titanium implementation also performs nonblocking array copy. This permits us to overlap communication during the global transpose with computation, giving us a second significant improvement over the Fortran code. As a result, the Titanium code performs 36% faster than Fortran on 64 processors of the Opteron/InfiniBand system.

7.3 MG Performance

For the MG benchmark, the Titanium code again uses nonblocking array copy to overlap some of the communication time spent in updating ghost cells. However, the performance benefit is not as great as for FT, since each processor can only overlap two messages at a time, and no computation is done during this time. Nonetheless, the results in figure 9 demonstrate that Titanium performs nearly identically to Fortran for both platforms and for both problem classes.

We also expect that the MG benchmark should scale nearly linearly, since the domain decomposition is load-balanced for all the graphed processor counts. In actuality, both implementations do scale linearly, as expected.

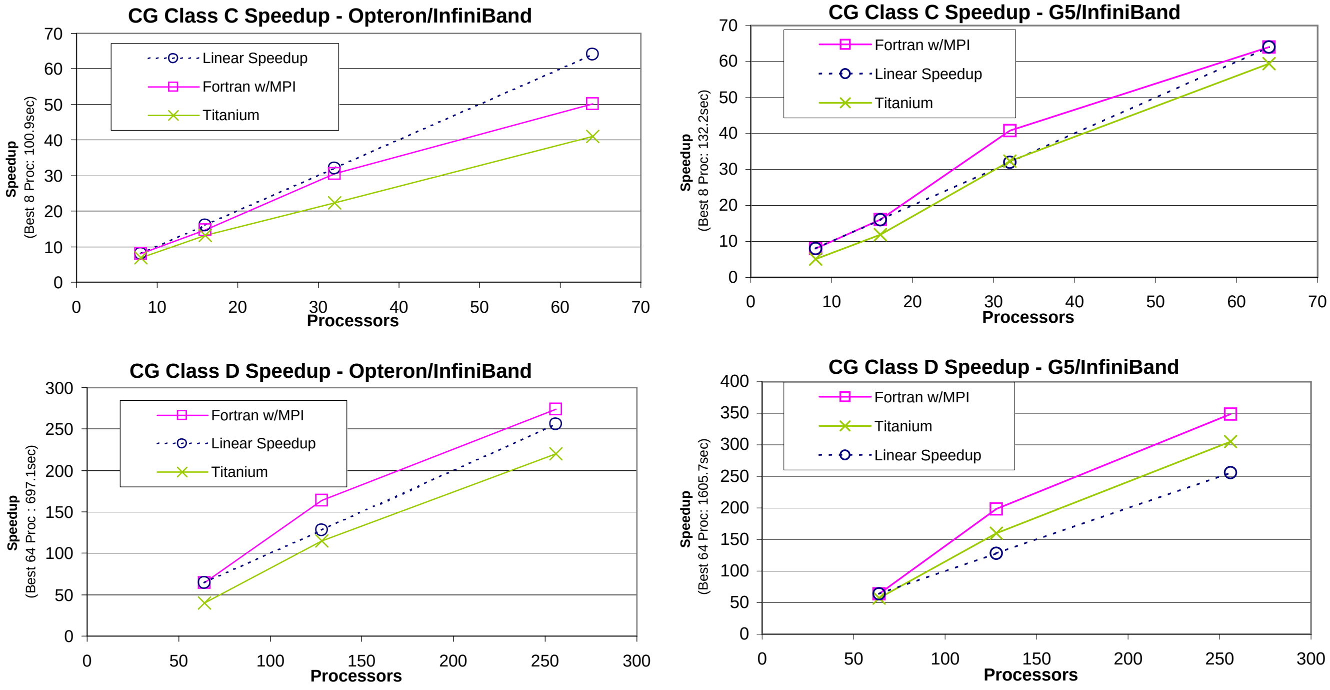

Figure 10. CG performance comparison

7.4 CG Performance

The Titanium CG code implements the scalar and vector reductions using point-to-point synchronization. This mechanism scales well, but only provides an advantage at larger numbers of processors. At small processor counts (8 or 16 on the G5), the barrier-based implementation is faster.

The CG performance comparison is shown in figure 10. In some cases the CG scaling for both Titanium and Fortran is super-linear due to cache effects – if a processor's portion of the sparse matrix does not fit into cache at a certain number of processors, it very well might with a higher processor count (and consequently a smaller portion of the matrix). Thus, in addition to the speedup resulting from processing a smaller portion of the matrix, cache misses in the local matrix-vector multiply are also reduced, resulting in super-linear speedups.

For both platforms, however, Titanium's performance is slightly worse than that of Fortran, by a constant factor of about 10-20%. One reason for this is that point-to-point synchronization is still a work in progress in Titanium. Currently, if a processor needs to signal to a remote processor that it has completed a put operation, it sends two messages. The first is the actual data sent to the remote processor. The second is an acknowledgment that the data has been sent. This will eventually be implemented as one message in Titanium, and should help bridge the remaining performance gap between the two languages.

7.5 Importance of Locality Qualification

Section 4.5 describes Titanium's *local* type qualifier, which can be used to statically annotate selected reference types in the program as always referring to data which resides locally. Local qualification enables several important optimizations in the implementation of pointer representation, dereference and array access that reduce serial overheads associated with global pointers and enable more effective optimization and code-generation by the backend C compiler. The Titanium optimizer includes a Local Qualification Inference (LQI) optimization that automatically propagates locality information gleaned from allocation statements and programmer annotations through the application code using a constraint-based inference [29]. Figure 11 illustrates the effectiveness of the LQI optimization by comparing the execution performance of the CG and MG implementations with the compiler's LQI optimization disabled or enabled, with identical application code. The graph demonstrates that in both cases there is a very significant benefit provided by the LQI optimization – by statically propagating locality information to pointer and array variables throughout the application, the optimization has effectively removed serial overheads associated with global pointers and delivered a total runtime speedup of 239% for CG and 443% for MG.

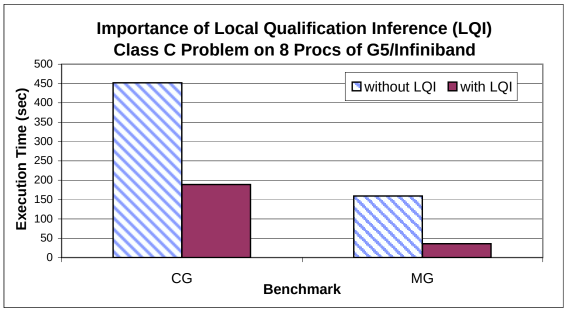

Figure 11. LQI compiler optimization speedup

8 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.

8.1 ZPL

ZPL is a data parallel language developed at the University of Washington. A case study by Chamberlain, Deitz and Snyder [12, 13] compared implementations of NAS MG across various machines and parallel languages (including MPI/Fortran, ZPL [41], Co-Array Fortran [36], High Performance Fortran [19], and Single-Assignment C). They compared the implementations in terms of running time, code complexity and conciseness. Our work extends theirs by providing a similar evaluation of Titanium for MG, but also includes 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 (namely, Titanium) need not suffer in conciseness, even when compared using benchmarks that are naturally expressed in the more restrictive data-parallel model – we see comparable code size between the two languages (192 lines total for MG 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 simple 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 more restrictive languages. Neither the performance nor the conciseness of Titanium appear suffer from this generality.

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 [20] and that of Nieplocha et al on ARMCI [35] 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 [37] and UPC [5] show that these performance advantages carry over to application-level performance in the NAS benchmarks.

8.2 Co-Array Fortran

Co-Array Fortran (CAF) is an explicitly parallel, SPMD, global address space extension to Fortran 90 initially developed at Cray Inc [36]. 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 [34]. 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 [14, 16, 37].

CAF has a built-in distributed data structure abstraction, but layouts are more restrictive than in a language like ZPL or HPF – because distribution is specified by identifying a co-dimension that is spread over the processors. Titanium's pointer-based layouts can be used to express arbitrary distributions. Communication is more visible in CAF than the other 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 MPI/Fortran MG code, and therefore not surprisingly demonstrated comparable length. In contrast, our Titanium implementations were written from scratch to best utilize the available language features and demonstrate the productivity advantages.

8.3 UPC

Unified Parallel C (UPC) [45] is a parallel extension of ISO C99 [10] 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 [7, 15] and Intrepid UPC [24] 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. [17] ported the MPI-Fortran versions of the NAS parallel benchmarks into UPC, with mixed performance results relative to MPI. Bell et al [5] 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 MPI-Fortran implementations.

Cantonnet et al. [11] studied the productivity benefits of UPC, based in part on line count analysis, concluding that the PGAS model and one-sided communication provide significantly improve programmer productivity by reducing conceptual complexity.

9 Conclusions

We described Titanium implementations and performance results for three of the NAS benchmarks, which reflect three different types of communication and computation patterns: nearest neighbor computation on a 3-D mesh (MG), FFT with an all-to-all transpose on a 3-D mesh (FT), and 2-D sparse matrices with indirect array accesses (CG). Titanium supports more general distributed data layouts and irregular parallelism patterns than are required for these problems. In addition, the use of Java as a base language provides support for strong typing, user-defined classes, inheritance, and dynamic memory management, all of which raise the level of abstraction when compared to most serial languages in common usage for parallel computing. Yet despite both these dimensions of additional language generality, Titanium proves to be well-suited to the NAS benchmarks from both an expressiveness and performance perspective.

Titanium's powerful multi-dimensional array abstraction and first-class language concepts for multidimensional indices (points) and domains make the 3-D codes particularly clean. The use of immutable classes as a general mechanism for extending the set of primitive types in the language is crucial: these concepts are used for the `Point` and `RectDomain` types that are essential to arrays, and the extensibility is shown in the user level `Complex` library used in FT. Although the Titanium compiler does not perform optimizations specialized to FFT computations on cache-based systems, the support for cross-language support provides the ability to expose a Titanium array to external libraries without data copy overheads, enabling the use of highly-tuned vendor libraries in Titanium applications. The one-sided communication model naturally provides both fine-grained access and expressive coarse-grained array copy operations, and was shown to achieve performance competitive with and sometimes noticeably faster than two sided message passing. Finally, while a shared address space managed in runtime software incurs some overhead in pointer size and access overhead, the Titanium language and optimizer support for identifying data structures that are local to a given processor were shown to be quite effective. Our Titanium implementations of the benchmarks are competitive with Fortran/MPI performance, and provide speedups of up to 36% in the case of FT. All of the Titanium codes scale well up to 256 processors using realistic problem sizes across several distributed memory machines.

Future work may improve on these results by experimenting with other benchmarks, and we believe there are opportunities for both language and compiler improvements motivated in this setting. For example, the sparse array data structures could be supported more directly if arrays could be built over non-rectangular domains, some of the communication optimizations performed in application code might be made automatic, and there are endless opportunities for serial performance optimizations. However, the results in this paper provide strong evidence that high level abstractions and programmer productivity need not be sacrificed to achieve high performance.

References

[1] A. Aiken and D. Gay. Barrier inference. In *Principles of Programming Languages, San Diego, California*, January 1998.

[2] A. Aiken and D. Gay. Memory management with explicit regions. In *Proceedings of Programming Language Design and Implementation, Montreal, Canada*, June 1998.

[3] 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.

[4] G. T. Balls and P. Colella. A finite difference domain decomposition method using local corrections for the solution of poisson's equation. In *Journal of Computational Physics, Volume 180, Issue 1, pp. 25-53*, July 2002.

[5] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing application performance using one-sided communication. Technical Report to appear, Lawrence Berkeley National Laboratory, October 2005.

[6] C. Bell, W. Chen, D. Bonachea, and K. Yelick. Evaluating Support for Global Address Space Languages on the Cray X1. In *19th Annual International Conference on Supercomputing (ICS)*, June 2004.

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

[8] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment, September 1988.

[9] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

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

[11] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *IPDPS*, 2004.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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".

[19] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS parallel benchmarks in high performance fortran. Technical Re-

port NAS-98-009, Nasa Ames Research Center, Moffet Field, CA, September 1998.

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

[21] E. Givelberg and K. Yelick. Distributed immersed boundary simulation in Titanium, 2003.

[22] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, second edition, 2000.

[23] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

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

[25] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *to appear in the proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC2005)*, 2005.

[26] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textual barriers. In *Submitted*, 2005.

[27] A. Krishnamurthy. *Compiler Analyses and System Support for Optimizing Shared Address Space Programs*. PhD thesis, U.C. Berkeley, 1998.

[28] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jorunal of Parallel and Distributed Computing*, 1996.

[29] B. Liblit and A. Aiken. Type systems for distributed data structures. In *the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2000.

[30] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *10th Annual International Static Analysis Symposium (SAS)*, June 2003.

[31] P. McCorquodale and P. Colella. Implementation of a multilevel algorithm for gas dynamics in a high-performance Java dialect. In *International Parallel Computational Fluid Dynamics Conference (CFD'99)*, 1999.

[32] S. Merchant. Analysis of a contractile torus simulation in Titanium, August 2003.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] G. Pike, L. Semenzato, P. Colella, and P. N. Hilfinger. Parallel 3d adaptive mesh refinement in Titanium. In *9th SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas*, March 1999.

[40] G. R. Pike. Reordering and storage optimizations for scientific programs, January 2002.

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

[42] J. Su and K. Yelick. Array prefetching for irregular array accesses in Titanium. In *Sixth Annual Workshop on Java for Parallel and Distributed Computing, Santa Fe, New Mexico*, April 2004.

[43] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[44] Titanium home page. http://titanium.cs.berkeley.edu.

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

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

[47] S. Yau, S. Merchant, and K. Yelick. Simulating blood flow in the heart with Titanium, a high-performance Java dialect, 2002.

[48] S. M. Yau. Experiences in using Titanium for simulation of immersed boundary biological systems, May 2002.

[49] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. In *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.

Appendix: Platforms on which Titanium was measured

System	Processor	Network	Software	Location
Opteron/ InfiniBand	Dual 2.2 GHz Opteron (320 nodes 4GB/node)	Mellanox Cougar InfiniBand 4x HCA	Linux 2.6.5, Mellanox VAPI, MVAPICH 0.9.5, Pathscale CC/F77 2.2	NERSC/Jacquard
G5/ InfiniBand	Dual 2.3 Ghz G5 (1100 nodes 4GB/node)	Mellanox Cougar InfiniBand 4x HCA	Apple Darwin 7.8.0, Mellanox InfiniBand OSX Driver v1.04, IBM XLC/XLF 6.0	Virginia Tech/SystemX