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

[1] Computer Science Division, University of California at Berkeley
[2] Lawrence Berkeley National Laboratory

Abstract. Titanium is an explicitly parallel dialect of Java^{TM} designed for high-performance scientific programming. We present an overview of the language features and demonstrate their use in the context of the NAS Parallel Benchmarks, a standard suite of common scientific kernels. We argue that parallel languages like Titanium provide greater expressive power than conventional approaches, enabling much more concise and expressive code that minimizes time to solution. Moreover, we have found that 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.

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.

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 [1], UPC [2] and Co-array Fortran [3] 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.

A major focus of this paper is to showcase the performance and productivity benefits of the Titanium programming language in the context of the NAS Parallel Benchmarks [4], a set of benchmarks representative of common scientific kernels. We demonstrate by example that scientific programming in PGAS languages like 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.

2 Titanium Overview

Titanium [1] is an explicitly parallel, SPMD dialect of Java^{TM} that provides a Partitioned Global Address Space (PGAS) memory model. 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 [5], it inherits all the expressiveness, usability and safety properties of that language.

Titanium notably adds a number of features to standard Java that are designed to support high-performance computing. They include: flexible and efficient multi-dimensional arrays, built-in support for multi-dimensional domain calculus, locality and sharing reference qualifiers, explicitly unordered loop iteration, user-defined immutable classes, operator-overloading, and cross-language support. These features are described in detail later in this paper, as well as in the Titanium language reference [6]. Titanium also adds several other features to Java, including: C++-style templates, user-controlled memory management with explicit memory zones, compile-time checking of barrier synchronization, and library support for synchronization and collective communication.

The current Titanium compiler implementation [7] uses a static compilation strategy - programs are translated to intermediate C code and then compiled to machine code using a vendor-provided C compiler. They are then 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. Thus, Titanium is extremely portable, and Titanium programs can basically be run unmodified on uniprocessors, shared memory machines and distributed memory machines. 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).

3 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks consist of a set of kernel computations and larger pseudo-applications taken primarily from computational fluid dynamics [4]. They reflect several different types of communication and computation patterns: nearest neighbor computation on a 3-D mesh (MG), FFTs with an all-to-all transpose on a 3-D mesh (FT), and 2-D sparse matrices with indirect array accesses (CG). However, 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 [8, 9].

The original reference implementation of the NAS Parallel Benchmarks is written in serial Fortran with MPI [10]. We use this implementation 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.

4 Titanium Features in the Multigrid (MG) Benchmark

4.1 Titanium Arrays

The NAS benchmarks, like many scientific codes, rely heavily on arrays for the main data structures. 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. 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 of 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);
}
```

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. Note that 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.

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:

```
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 ] );
}
```

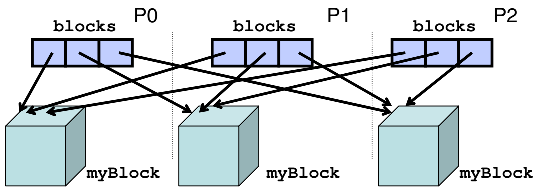

Fig. 1. Distributed data structure built using the `exchange` operation in MG

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 [11].

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 [11]. 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 [8]. 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];

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

First, each processor computes its start and end indices by performing point arithmetic operations. These indices are used to create the local 3-D array `myBlock`. Then, the pointer-based distributed data structure `blocks` is created using the `exchange` collective. Figure 1 illustrates the resulting data structure for a 3-processor execution.

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. Ghost cells are a set of array elements surrounding the local grid that cache elements of neighboring grids. Simple array operations can be used to fill in these ghost regions, thereby migrating 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. 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 benchmark requires periodic boundary conditions, and an additional array view operation is needed 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 asking 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.

The Titanium optimizer also includes a Local Qualification Inference (LQI) optimization that automatically propagates locality information gleaned from allocation statements and programmer annotations in the application code using a constraint-based inference [12]. LQI can effectively remove serial overheads associated with global pointers, as evidenced by the 81% reduction in the running time of MG on 8 processors of the G5/InfiniBand machine.

4.6 The MG Benchmark Implementation

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.

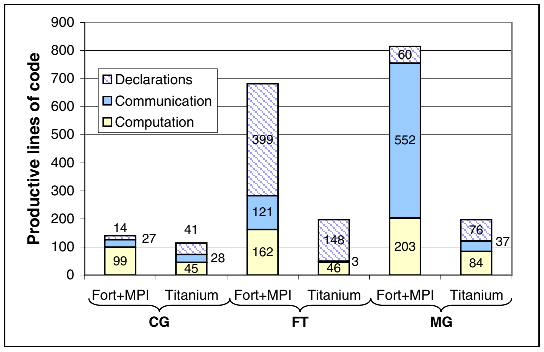

Fig. 2. Timed region line count comparison

The figure shows that MG communication and computation line counts heavily favor Titanium. This discrepancy is mainly due to Titanium's domain calculus and array copy operations, and to a lesser extent, Titanium array features for local stencil computations.

5 Titanium Features in the Conjugate Gradient (CG) Benchmark

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 [13, 14]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

Another example 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, so the `rowRectDomains` array contains a *RectDomain* for each row of the matrix. Each *RectDomain* then contains its row's first and last indices for arrays `colIdx` and `a`.

```
// 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;
} }
```

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 The CG Benchmark Implementation

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. 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.

6 Titanium Features in the Fourier Transform (FT) Benchmark

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 3 compares how one might define a Complex number class using either standard Java Objects versus Titanium immutables.

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;
```

Fig. 3. Complex numbers in Java and Titanium

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.

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. 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 [15] 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.

6.3 Nonblocking Arraycopy

Titanium's explicitly nonblocking array copy library methods helped in implementing a more efficient 3-D FFT.

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. 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-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. Reorganizing the communication in FT to maximize overlap results in a large performance gain, as seen in figure 4.

6.4 The FT Benchmark Implementation

In terms of code size, 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 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

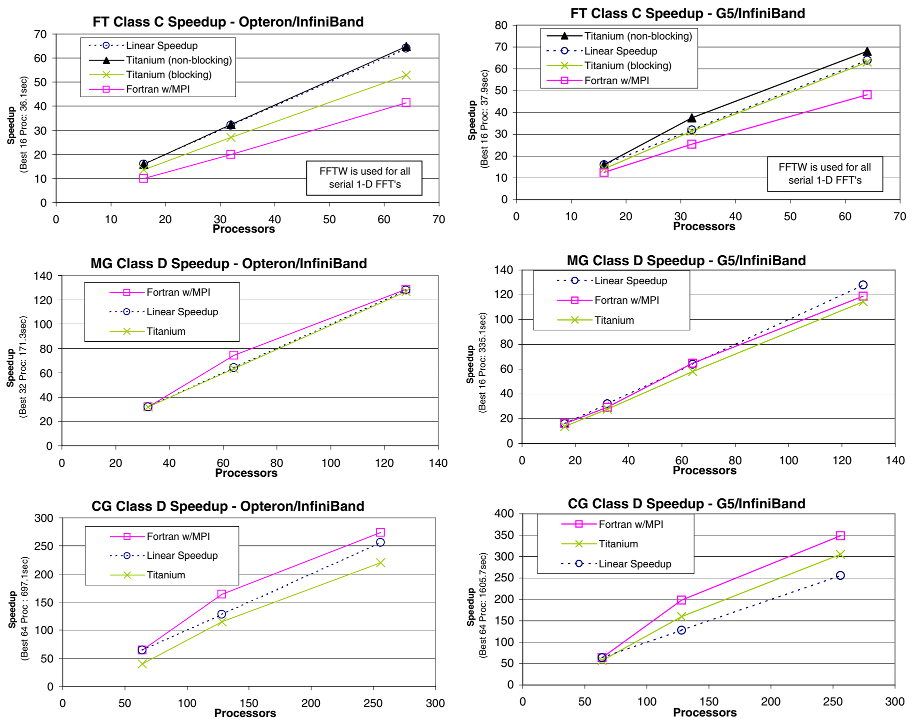

Fig. 4. Performance comparisons for FT, MG, and CG respectively

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 tested the Titanium and Fortran with MPI implementations on an Opteron cluster and a G5 cluster, both with InfiniBand interconnects. For details concerning the input sizes for each problem class, please see the NAS benchmark specification [4].

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.

The actual performance results for all three benchmarks are shown in figure 4. 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. Consequently, the language that has the higher speedup for a given number of processors *actually runs faster* for that case.

7.2 FT Performance

Both implementations of the FT benchmark use the same version of the FFTW library [15] for the local 1-D FFT computations, since it always outperformed 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 at the top of figure 4, 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 4 demonstrate that Titanium performs nearly identically to Fortran for both platforms and for both problem classes.

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 at the bottom of figure 4. In some cases the CG scaling for both Titanium and Fortran is super-linear due to cache effects. 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, and 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.

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.

ZPL is a data parallel language developed at the University of Washington. A case study by Chamberlain, Deitz and Snyder [11] compared implementations of NAS MG across various machines and parallel languages (including MPI/Fortran, ZPL, Co-Array Fortran [3], High Performance Fortran, 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.

Co-Array Fortran (CAF) is an explicitly parallel, SPMD, global address space extension to Fortran 90 initially developed at Cray Inc [3]. CAF has a built-in distributed data structure abstraction. However, layouts are more restrictive than in a language like ZPL or HPF, since 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).

Unified Parallel C (UPC) [2] is a parallel extension of ISO C99 that provides a global memory abstraction and communication paradigm similar to Titanium. The Berkeley UPC [16] and Intrepid UPC 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. Bell et al [17] reimplemented some of the NAS benchmarks in UPC's one-sided communication paradigm, producing performance improvements of up to 2x over the MPI-Fortran versions.

9 Conclusions

We have shown that Titanium is well-suited to three common yet diverse scientific kernels from both an expressiveness and performance standpoint. However, Titanium applications are not merely limited to the NAS benchmarks, as it supports more general distributed data layouts and irregular parallelism patterns than these problems require. 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 these features help raise the level of abstraction when compared to most serial languages commonly used in parallel computing.

References

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

[2] UPC Community Forum: UPC specification v1.2. (2005) http://upc.gwu.edu/documentation.html.

[3] Numrich, R., Reid, J.: Co-array fortran for parallel programming. In: ACM Fortran Forum 17, 2, 1-31. (1998)

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

[5] Gosling, J., Joy, B., Steele, G.: The Java Language Specification. second edn. (2000)

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

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

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

[9] Givelberg, E., Yelick, K.: Distributed immersed boundary simulation in Titanium (2003)

[10] 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.

[11] Chamberlain, B.L., Deitz, S.J., Snyder, L.: 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)

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

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

[14] Pike, G.R.: Reordering and storage optimizations for scientific programs (2002)

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

[16] The Berkeley UPC Compiler (2002). http://upc.lbl.gov.

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