Parallel Languages and Compilers: Perspective from the Titanium Experience*

Katherine Yelick[1,2], Paul Hilfinger[1], Susan Graham[1], Dan Bonachea[1],
Jimmy Su[1], Amir Kamil[1], Kaushik Datta[1], Phillip Colella[2], and Tong Wen[2]
{*yelick, hilfingr, graham, bonachea, jimmysu, kamil, kdatta*}*@cs.berkeley.edu*
{*pcolella, twen*}*@lbl.gov*
Computer Science Division, University of California at Berkeley[1]
Lawrence Berkeley National Laboratory[2]

August 16, 2005

Abstract

We describe the rationale behind the design of key features of Titanium—an explicitly parallel dialect of Java™ for high-performance scientific programming—and our experiences in building applications with the language. Specifically, we address Titanium's global address-space model, memory management, SPMD parallelism support, multi-dimensional arrays and array-index calculus, immutable classes (class-like types that are value types rather than reference types), operator overloading, and generic programming. We summarize results and lessons learned from implementing the NAS parallel benchmarks, elliptic and hyperbolic solvers using AMR, and several applications of the immersed boundary method.

1 Introduction

Titanium is an explicitly parallel dialect of Java™ designed for high-performance scientific programming [60]. The Titanium project started in 1995, at a time when custom supercomputers were losing market share to PC clusters. The motivation was to create a language design and implementation enabling portable programming for parallel platforms that strikes an appropriate balance between expressiveness, user provided information about possibly concurrent execution, and compiler and runtime support for parallelism. Our goal was to design a language that could be used for high performance on some of the most challenging applications, such as those with adaptivity in time and space, unpredictable dependencies, and sparse, hierarchical, or pointer-based data structures.

The strategy we used was to build on the experience of several global address space languages, including Split-C [18], CC++ [32], and AC [16], but to design a higher-level language offering object-orientation with strong typing and safe memory management in the context of applications requiring high-performance and scalable parallelism. Although Titanium initially used C++ as a base language, there were several reasons why there was an early decision to design Titanium as a dialect of Java instead. Relative to C++, Java is a semantically simpler and cleaner language, making it easier to extend. Also, Java is a type-safe language, that protects programmers from the obscure errors that can result from violations of unchecked runtime constraints. Type-safety enables users to write more robust programs and the compiler to perform better optimizations. Java has also become a popular teaching language, providing a growing community of users for whom the basics of Titanium should be easy to master.

The standard Java language alone is insufficient for large-scale scientific programming. Its multi-dimensional array support makes heavy use of pointers, and is fundamentally asymmetric in its treatment of the dimensions. Its memory model is completely flat, making no provision for distributed or otherwise hierarchical memories. Its multi-processing support does not distinguish "logical threads," used as program-structuring devices and intended to operate sequentially, from "process-like threads," intended to represent opportunities for concurrency. This conflation impacts static program analysis required by some optimizations.

It is possible to approach these deficiencies either through language extensions or library extensions. The former choice allows more concise and user-friendly syntax, and makes more information explicitly available to the compiler. The latter choice would perforce be more portable. However, it was clear that in either case, we would have to modify or build a compiler to get the necessary performance, and that while the library-only approach would be portable in a purely functional sense, it would make portability of application performance more problematic. For these reasons, we chose to introduce a new dialect. We argue that

*This work was supported in part by the Department of Energy under DE-FC03-01ER25509, by the California State MICRO Program, by the National Science Foundation under ACI-9619020 and EIA-9802069, by the Defense Advanced Research Projects Agency under F30602-95-C-0136, and by Sun Microsystems. Machine access was provided by NERSC/DOE, SDSC/NSF, PSC/NSF, LLNL/DOE, U.C. Berkeley, Virginia Tech, and Rice University. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

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.

In the remainder of the paper, we present highlights of the design of the Titanium language, our experience using it for scientific applications, and compilation and runtime innovations that support efficient execution on sequential and parallel platforms.

2 Models of Parallelism

Titanium is based on a static Single Program Multiple Data (SPMD) model of parallelism, in which a fixed number of threads are created at program startup and remain throughout the execution. The parallelism is *explicit* in the language semantics, in the sense that a serial, deterministic abstract machine cannot describe all the behaviors that a Titanium program may produce. The SPMD model offers more flexibility than an implicit model based on data parallelism or automatic parallelization of serial code, and more user control over performance than either data-parallel or general task-parallel approaches.

2.1 Data Parallelism

Data-parallel languages like ZPL, NESL, HPF and pC++ are popular as research languages because of their semantic simplicity: the degree of parallelism is determined by the data structures in the program, and need not be expressed directly by the user [10, 11, 27, 53]. These languages include array operators for elementwise arithmetic operations, e.g., $C = A+B$ for matrix addition, as well as reduction and scan operations to compute values such as sums over arrays. In their purest form, data-parallel languages are implicitly parallel, so their semantics can be defined serially: assignment statements are defined by evaluation of the entire right-hand side before any modifications to left-hand side variables are performed and there are implicit barriers between statements.

The semantic simplicity of data-parallel languages is attractive, yet these languages are not widely used in practice today. While the factors in language success involve complex market and sociological factors, there are two technical problems that have limited the success of data-parallel languages as well: 1) They are not expressive enough for some of the most irregular parallel algorithms; 2) They rely on fairly sophisticated compiler and runtime support that takes control away from applications programmers. We describe each of these issues in more detail and how solutions to address the first tend to trade off against the second.

The data-parallel model is fundamentally limited to performing identical operations in parallel, which makes computations like divide-and-conquer parallelism or adaptivity challenging at best. NESL generalizes the model to include nested data structures with nested parallelism, but complex dependence patterns such as those arising in parallel discrete-event simulation or sparse direct methods are still difficult to express. HPF goes even further by adding the *INDEPENDENT* keyword, which can be used for general (not just data-parallel) computation. Rather than starting with a restricted data-parallel language and augmenting it to support these applications, Titanium starts with the more general SPMD model to address the needs of irregular applications at the outset.

The second problem with data-parallel languages is the lack of control over parallel resources, since the logical level of parallelism in the application is likely many times larger than the physical parallelism available on the machine. On massively parallel SIMD machines of the past, the mapping of data parallelism to processors was straightforward, but on modern machines built from heavyweight processors (either general-purpose microprocessors or vector processors), the compiler and runtime system must map the fine-grained parallelism onto coarse-grained machines. HPF and ZPL both provide data-layout primitives so that the user can control the mapping of data to processors, but the decomposition of parallel work must still be derived by the language implementation from these layout expressions. This work-decomposition problem proved to be quite challenging for complex data layouts or for the case when multiple arrays with different distributions are involved.

2.2 Task Parallelism

At the opposite extreme from data-parallel languages are task-parallel languages, which include the Java thread model as well as languages extended with threading libraries such as POSIX [28]. These models allow programmers to express parallelism between arbitrary computations, so they can be used for the most complicated sorts of parallel dependence patterns, but they still lack direct user control over parallel resources. The parallelism unfolds at runtime, so it is normally the responsibility of the runtime system to control the mapping of threads to processors. To avoid the overhead of context switching and thread scheduling, scientific applications written with a task-parallel model typically start a fixed number of threads equal to the number of processors.

2.3 Titanium's Parallelism Model

Titanium's SPMD parallelism model is familiar to users of message-passing models such as MPI [41]. The following example shows a simple Titanium program that illustrates the use of built-in methods `Ti.numProcs()` and `Ti.thisProc()`, which query the environment for the number of threads (or processes) and the index within that set of the executing thread. The example prints these indices in arbitrary order. (Note that many systems ensure that interleaving from different threads happens on a line-by-line basis but in some cases and on some spawners partial lines might be interleaved with each other). The number of Titanium threads need not be equal to the number of physical processors, a feature that is often useful when debugging parallel code on single-processor machines. However, high-performance runs typically use a one-to-one mapping between Titanium threads and physical processors.

```
class HelloWorld {
   public static void main (String [] argv) {
      System.out.println("Hello from proc " + Ti.thisProc() + " out of " + Ti.numProcs());
   }
}
```

Titanium supports Java's synchronized blocks, which are useful for protecting asynchronous accesses to shared objects. Because many scientific applications use a bulk-synchronous style, we also added a barrier-synchronization construct, `Ti.barrier()`, as well as a set of collective communication operations to perform broadcasts, reductions, and scans. A novel feature of Titanium's parallel execution model is that barriers must be textually aligned in the program—not only must all threads reach a barrier before any one of them may proceed, but they must all reach *the same textual* barrier. For example, the following program is not legal in Titanium:

```
if (Ti.thisProc() == 0)
   Ti.barrier();  // illegal barrier
else
   Ti.barrier();  // illegal barrier
```

Aiken and Gay developed the static analysis the compiler uses to enforce this alignment restriction, based on two key concepts [1]:

- A *single method* is one that must be invoked by all threads collectively. Only single methods can contain barriers.
- A *single-valued expression* is an expression that is guaranteed to evaluate to the same value on all processes. Only single-valued expressions may be used in conditional expressions that affect which barriers or single-method calls get executed.

The compiler automatically determines which methods are single by finding barriers or (transitively) calls to other single methods. Single-valued expressions are required in statements that determine the flow of control to barriers, ensuring that the barriers are executed by all threads or by none. Titanium extends the Java type system with the `single` qualifier. Variables of single-qualified type may only be assigned values from single-valued expressions. Literals and values that have been broadcast are simple examples of single-valued expressions. The following example illustrates these concepts. Because the loop contains barriers, the expressions in the for-loop header must be single-valued, which the compiler can check statically, since the variables are declared single and are assigned from single-valued expressions.

```
int single allTimestep = 0;
int single allEndTime = broadcast inputTimeSteps from 0;
for (; allTimestep < allEndTime; allTimestep++){
  < read values belonging to other threads >
  Ti.barrier();
  < compute new local values >
  Ti.barrier();
}
```

Barrier analysis is entirely static and provides compile-time prevention of barrier-based deadlocks. It can also be used to improve the quality of concurrency analysis used in optimizations as we will show. Single qualification on variables and methods is a useful form of program design documentation, improving readability by making replicated quantities and collective methods explicitly visible in the program source and subjecting these properties to compiler enforcement. Our compiler includes an optional single-method inference that can be used as a convenience to automatically infer which methods must be single-qualified to maintain the correctness properties for collective operations. However, our experience is that the use of inference can sometimes produce errors whose cause is obscure, as when the analysis detects that activating an exception on some threads might cause them to bypass a barrier or to fail to update a single variable properly.

3 Models of Communication and Sharing

The two basic mechanisms for communicating between threads are accessing shared variables and sending messages. Shared memory is generally considered easier to program, because communication is one-sided: threads can access shared data at any time without interrupting other threads, and shared data structures can be directly represented in memory. As a programming model, the term "shared memory" normally refers to a uniform memory-access-time abstraction, which means that data is cached locally, and so can be accessed quickly after the first access. Message passing is more cumbersome, requiring both a two-sided protocol and packing and unpacking for non-trivial data structures. It is also more popular than shared memory on large-scale machines because it makes data layout explicit and is easily implemented on loosely coupled systems lacking complex distributed cache-coherence hardware.

Titanium is based on a *Partitioned Global Address Space (PGAS)* model, which is similar to shared memory but without the uniform access time assumption. As shown in Figure 1, memory is partitioned such that each partition has *affinity* to one thread. Memory is also partitioned orthogonally into private and shared memory, with stack variables living in private memory, and heap objects, by default, living in the shared space. A thread may access any variable that resides in shared space, but has fast access to variables in its own partition. Objects created by a given thread will reside in its own part of the memory space.

Titanium statically makes an explicit distinction between *local* and *global* references: a local reference must refer to an object within the same thread partition, while a global reference may refer to either a remote or local partition. In Figure 1, instances of `l` are local references, whereas `g` and `nxt` are global references and can therefore cross partition boundaries. The motivation for this distinction is performance. Global references are more general than local ones, but they often incur a space penalty to store affinity information and a time penalty upon dereference to check whether communication is required. References in Titanium are global by default, but may be designated local using the `local` type qualifier, which is discussed in section 4.6.

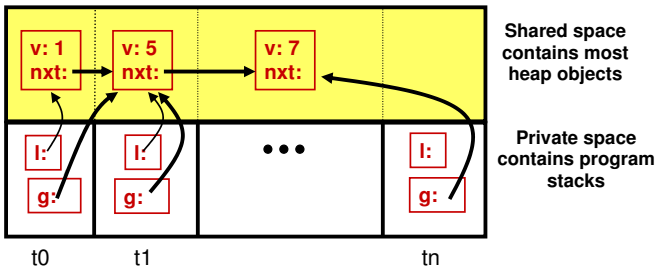

Figure 1: Titanium's Memory Model.

The partitioned memory model is designed to scale well on distributed memory platforms without the need for caching of remote data and the associated coherence protocols. Titanium also runs well on shared-memory multiprocessors and uniprocessors, where the partitioned-memory model may not correspond to any physical locality on the machine and the global references generally incur no overhead relative to local ones. Naively-written Titanium programs may ignore the partitioned-memory model and, for example, allocate all data structures in one thread's shared memory partition or perform fine-grained accesses on remote data. Such a program would run correctly on any platform but would likely perform poorly on a distributed memory platform. In contrast, a program that carefully manages its data-structure partitioning and access behavior in order to scale well on distributed memory hardware is likely to scale well on shared-memory platforms as well. The partitioned model provides the ability to start with a functional, shared-memory style code and incrementally tune performance for distributed memory hardware by reorganizing the affinity of key data structures or adjusting access patterns in program bottlenecks to improve communication performance.

4 Titanium Extensions to Java

We added several features to Java to better support scientific computation and high performance. In this section we illustrate these features, drawing on examples taken from our implementatins of the three NAS Parallel Benchmarks [4, 19]: Conjugate Gradient (CG), 3D Fast Fourier Transform (FT), and Multigrid (MG). These benchmarks, like most scientific applications, rely heavily on multidimensional arrays as their primary data structures: CG uses simple 1D arrays to represent vectors and a set of 1D arrays to represent a sparse matrix, while both MG (Multigrid) and FT (Fourier Transform) use 3D arrays to represent a discretization of physical space. These NAS benchmarks are sufficient for illustrating Titanium features, but some of the generality was motivated by more complicated parallel computations, such as Adaptive Mesh Refinement [57] and Immersed Boundary Method simulation [47], which are more extensive application studies described in Section 5.

4.1 Titanium Arrays

In Java, all arrays are objects and only 1D arrays are fully supported. Multidimensional arrays are represented as arrays of arrays. While this approach is general, it incurs performance penalties from the extra level of indirection, the memory layout, and the added complexity of compiler analysis. Therefore, iterating through any array with dimensionality greater than one is likely to be slow. Since MG, FT, and AMR all require 3D arrays, these applications would likely not perform well in standard Java, without converting all the arrays into 1D arrays and using tedious manual indexing calculations to emulate multidimensionality.

Titanium extends Java with a powerful multidimensional array abstraction, which provides the same kinds of sub-array operations available in Fortran 90. Titanium arrays are indexed by *points* and built on sets of points, called *domains*. Points and domains are first-class entities in Titanium – they can be stored in data structures, specified as literals, passed as values to methods and manipulated using their own set of operations. For example, the class A version of the MG benchmark requires a 256^3 grid with a one-deep layer of surrounding ghost cells, resulting in a 258^3 grid. Such a grid can be constructed with the following declaration:

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

The 3D Titanium array `gridA` has a rectangular index set that consists of all points $[i, j, k]$ with integer coordinates such that $-1 \leq i, j, k \leq 256$. Titanium calls such an index set a *rectangular domain* of 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. Programmers familiar with C or Fortran arrays are free 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 ability to specify points as named constants can be used to write stencil operations such as those used for MG. The following code applies a 5-point 2D stencil to each point `p` in `gridAIn`'s domain,where `gridAIn` denotes that portion of `gridA` that excludes the ghost regions (see Section 4.4), and then writes the resulting value to the same point in `gridB`.

```
final Point<2> EAST =  [ 1, 0];
final Point<2> WEST =  [-1, 0];
final Point<2> NORTH = [ 0, 1];
final Point<2> SOUTH = [ 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 NAS MG code used for benchmarking in Section 5.4 includes a 27-point stencil applied to 3D arrays. The Titanium code, like the Fortran code for this benchmark, uses a manually-applied stencil optimization that eliminates redundant common subexpressions [17]. The `foreach` construct is explained in the next section.

4.2 Foreach Loops

Titanium provides an unordered looping construct, *foreach*, specifically designed for iterating through a multidimensional space. In the `foreach` loop below, the point `p` plays the role of a loop index variable. (The stencil operation above has been abstracted as a method `applyStencil`).

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

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.

Note that this one loop concisely expresses an iteration over a multidimensional domain that would correspond 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 automatically manage the iteration boundaries for the multidimensional traversal.

In addition, 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 instance, by performing automatic cache blocking and tiling optimizations [48, 50]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

4.3 Distributed Arrays

Titanium also supports the construction of distributed array data structures in the Partitioned Global Address Space. Since distributed data structures are built from local pieces rather than declared as distributed types, Titanium is referred to as a "local view" language in the terminology of Chamberlain *et al.* [17]. 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 `blocks3D` distributed array, which can access any processor's portion of the grid. By convention, `myBlock` refers to the block in the processors partition (i.e., the local block).

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

double [3d] myBlock = new double[startCell:endCell];

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

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

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

First, each processor computes its start and end indices by performing arithmetic operations on Points. These indices are used to create a local `myBlock` array. Every processor also allocates its own 1D array `blocks`. Then, by combining the `myBlock` arrays using the `exchange` operation, `blocks` becomes a distributed data structure. As shown in Figure 2, the `exchange` operation performs an all-to-all broadcast and stores each processor's contribution in the corresponding element of its local `blocks` array.

Now `blocks` is a distributed data structure, but it maps a 1D array of processors to blocks of a 3D grid. To create a more natural mapping, a 3D array called `blocks3D` is introduced. It uses `blocks` and a method called `procForBlockPosition` (not shown) to establish an intuitive mapping from a 3D array of processor coordinates to blocks in a 3D grid. Both the block and cell positions are in global coordinates.

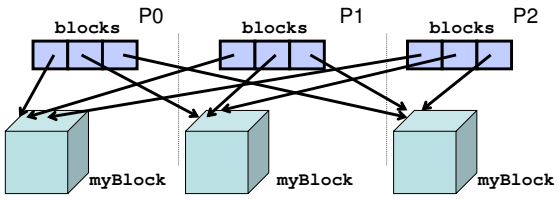

Figure 2: Distributed data structure created by Titanium's `exchange` operation for three processors.

In comparison with 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, the generality of Titanium's distributed data structures is not fully utilized in the NAS benchmarks, because the data structures are simple distributed arrays, rather than trees, graphs or adaptive structures. Titanium's pointer-based data structures can be used to express a set of discontiguous blocks—as in the AMR code described in section 5.1—or an arbitrary set of objects; they are 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

The true power of Titanium arrays stems from array operators that can be used to create alternative views of an array's data, all without an implied copy of the data. While this is useful in most scientific codes, it is especially valuable in hierarchical grid algorithms like Multigrid and AMR. Since these algorithms typically deal with many kinds of boundary operations for ghost cell updates, subarrays are usually required. Java does not handle such applications well, due to its non-contiguous memory layout. Even C and C++ do not support subarrays well, and hand-coding can often confuse the compiler.

Titanium's domain calculus operators, however, support subarrays both syntactically and from a performance standpoint. The tedious business of index calculations and array offsets has been migrated from the application code to the compiler and runtime system. For example, in the MG benchmark, the entire Titanium code for updating one plane of ghost cells between two adjacent processor blocks is as follows:

```
// use interior as in stencil code
double [3d] myBlockIn = myBlock.restrict(myBlock.domain().shrink(1));

// update overlapping ghost cells of neighboring block
blocks[neighborPos].copy(myBlockIn);
```

The first statement creates a new array variable, `myBlockIn`, that shares all of its elements with the non-ghost cells of `myBlock`. In general, the `restrict` operation creates an array view whose index set is restricted to the points in the domain argument. In the example above, this domain is computed by shrinking the index set of `myBlock` by one element on each side.

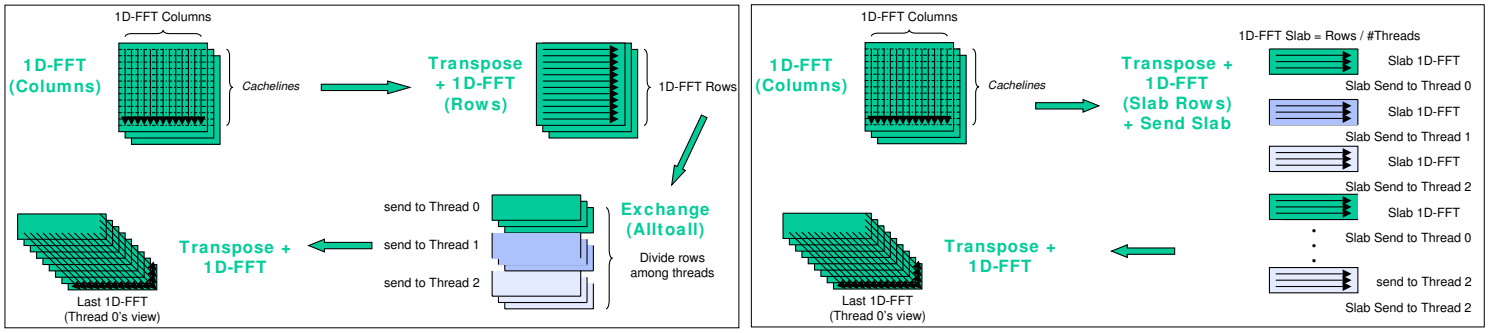

Figure 3: Two 3D FFT communication patterns in NAS FT – the left side shows the blocking bulk-synchronous exchange algorithm used by the stock Fortran w/MPI code, while the right side shows the nonblocking overlapped slabs algorithm used by Titanium.

Thus, `myBlockIn` refers to non-ghost elements of `myBlock`. The same could also be accomplished with a literal expression of the selected `RectDomain`, e.g.: `myBlock.restrict([[0,0,0]:[255,255,255]])`.

The second statement performs actual communication. 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 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 edge of the problem domain. At these locations, the MG benchmark uses periodic boundary conditions. To handle them properly, an additional array view operation is required to logically translate 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 above translates the indexes of the array view, creating a new view where the relevant points overlap their corresponding non-ghost cells in the subsequent copy.

4.5 Non-blocking Array Copy

Although the array copy operation is conceptually simple, it can be an expensive operation when it implies communication on distributed memory machines. Titanium enables the programmer to indicate when the communication induced by `copy` can be overlapped with independent computation or other communication, by selecting the `copyNB` library method to initiate non-blocking copying, and later ensure completion of the asynchronous communication using a second library call.

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

Both these issues can be dealt with using a slight reorganization of the 3D FFT algorithm employing non-blocking array copy. The new algorithm, implemented in Titanium, first performs a local 1D FFT, followed by a local transpose and a second 1D FFT. However, unlike the bulk-synchronous model, we begin sending each processor's portion of the grid (a 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 non-blocking array copy, addressing the other issue with the original algorithm. Non-blocking 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 (Section 6.3) 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 6.

4.6 The Local Keyword and Locality Qualification

The `blocks` distributed array in Figure 2 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 prefetched 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 advising 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 includes a Local Qualification Inference (LQI) optimization. Using a constraint-based inference, it automatically propagates locality information gleaned from allocation statements and programmer annotations through the application code [34]. 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. Figure 4 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 benchmarks significant benefit is 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.

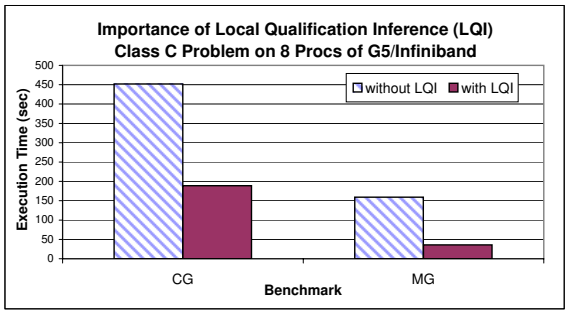

Figure 4: Performance speedup obtained with the LQI compiler optimization

The distinction between local and global references is modeled after Split-C, but Split-C pointers are local by default, whereas Titanium references are global by default. The global default makes it easier to port shared memory Java code into Titanium, since only the parallel thread creation needs to be replaced to get a functional parallel Titanium code. Split-C's local default discourages the use of gratuitous global pointers, so that local qualification inference is less likely to be needed.

4.7 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. Immutables 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. If one were to define a Complex class using standard Java Objects, it might include code something like this:

```
public class Complex {
  private double real;
  private double imag;
  public Complex(double r, double i) { real = r; imag = i; }
  public double getReal { return real; }
  public double getImag { return imag; }
  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);  }
  ...
}

Complex c = new Complex(7.1, 4.3);
Complex c2 = c.add(c).multiply(14.7);
```

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. 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).

A Titanium-based implementation of Complex using immutables and operator overloading is available in the Titanium standard library and includes code like this:

```
public immutable class Complex { // <-- note addition of 'immutable'
  public double real;
  public double 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);  }
  ...
}

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

Immutable types are not subclasses of java.lang.Object, and induce no overheads for pointers or Object headers. In addition, 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 example above 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 for immutables). Overloading 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).

4.8 Memory Management

In object-oriented languages, dynamic memory management is both a source of bugs (because memory is freed too soon) and a source of performance inefficiencies (because memory is freed too late). One marked contrast between C++ and Java is in their approaches to memory management: in C++, memory de-allocation is the programmer's responsibility, and in Java it is the runtime system's (specifically, the garbage collector's). As a result, a significant portion of the semantic complexity in C++

is devoted to giving programmers devices with which to build memory allocators, and memory management issues occupy a significant portion of programmers' attention.

As a dialect of Java, Titanium uses garbage collection. In order to minimize our implementation effort and to achieve portability, we chose to adopt the widely used Boehm-Demers-Weiser conservative garbage collector [12, 13] for collection within a single shared memory, rather than implementing our own collector. Conservative collectors do not depend on type information from the compiler to find pointers, but instead use a simple heuristic, "if it looks like a valid pointer, assume that it is," to mark a superset of reachable storage as active. They do not collect all garbage and spend some unnecessary time scanning data that does not contain pointers, but have proven to be effective at a reasonable cost (Detlefs, Dosser, and Zorn reported a 20% execution-time penalty over a variety of non- scientific benchmarks in C, compared to explicit allocation and deallocation [20]). For distributed collection, we add to this a simple conservative scheme in which local pointers that are communicated to a remote node are added to the local root set. This scheme is sound (collects no active storage), but leaky, since once an object becomes known outside a node, it is never collected.

We considered a more accurate distributed collection model, but were concerned about scalability of any fully automatic solution, and the required development effort, since we use several platforms where the Boehm-Demers-Weiser collector is not supported. We desired a mechanism that would give programmers some control over memory management costs as well as locality within a cache-based memory system, but without sacrificing safety. To this end, we added a region-allocation facility to Titanium, using the work of Gay and Aiken [24]. A programmer may create objects that serve as *regions* of memory to be used for allocation, and may then specify in which region any heap-allocated object is to be placed. All allocations in a region may be released with a single method call. Regions constitute a compromise – they require some programming effort, but are generally easier to use than explicit object-by-object de-allocation. They also represent a safety compromise: deleting a region while it still contains live objects is an error, which our implementation might not detect. Because programmers typically delete regions at well-defined major points in their algorithms, this danger is considerably reduced relative to object-by-object de-allocation.

One other problem with regions indicates an area in which our design needs refinement. From the programmer's point of view, many data structures involve hidden memory allocations. The built-in type `Domain` uses internal linked structures, for example, so that innocent-looking expressions involving intersections or unions may actually allocate memory or cause structure to be shared. Controlling the regions in which this happens (while possible) is often clumsy and error-prone. The overall lesson from our experiences is that although our compromises have been effective in allowing interesting work to get done, a production implementation would probably need a true, appropriately specialized garbage collector.

4.9 Cross-Language Calls

One of the hallmarks of scientific codes is the use of well-debugged and well-tuned libraries. 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.

As further explained in Section 6.1, Titanium is implemented as a source-to-source translator to C. This means that any library offering a C interface is potentially callable from Titanium. Since 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.

For example, the Titanium NAS FT implementation calls the FFTW library [22] to perform the local 1D FFT computations, thereby leveraging its auto-tuning features and machine-specific optimizations. Note that although the FFTW library does offer a 3D MPI-based parallel FFT solver, our benchmark only uses the serial 1D FFT kernel – Titanium code is used to create and initialize all the data structures, as well as to orchestrate and perform all the interprocessor communication.

One of the challenges of the native code integration with FFTW was manipulating the 3D Titanium arrays from within native methods, where their representation as 1D C arrays is exposed to the native C code. This was a bit cumbersome, especially since the FT implementation intentionally includes padding in each row of the array to avoid cache-thrashing. However, it was only because of Titanium's support for true multidimensional arrays that such a library call was even possible, since the 3D array data is stored natively in a row-major, contiguous layout. Java's layout of "multidimensional" arrays as 1D arrays of pointers to 1D arrays implies discontiguity of the array data that would have increased significantly the computational costs and complexity associated with calling external multidimensional computational kernels like FFTW.

4.10 Templates

In its original version, Titanium lacked any facility for generic definitions (*templates* to the C++ programmer), but we quickly saw the need for them. A minor reason was the syntactic irregularity of having predefined parameterized types such as `Point<3>` in

[1] Our use of a conservative garbage collector for automatic memory management eliminates the need to statically identify the location of all pointers at runtime, which eases interoperability with external libraries relative to copying garbage collectors which are typically used by standard Java implementations.

the library with no mechanism for programmers to introduce more. The major reason, however, came directly from applications. Here, generic types have many uses, from simple utility data structures (`List<int>`) to elaborate domain-specific classes such as distributed AMR grid structures in which the type parameter encapsulates the state variables.

Titanium's formulation of generic types long predates their introduction into Java with the 5.0 release in August, 2004. Partially as a result of that, our design differs radically from that of Java. The purely notational differences are superficial (Titanium uses a syntax reminiscent of C++), but the semantic differences are considerable, as detailed in the following paragraphs.

Values as generic parameters. As in C++, but unlike Java, generic parameters in Titanium may be constant expressions as well as types (providing the ability to define new types that are like `Point<3>`). To date, this feature has not seen much use outside of the built-in types. In principle, one could write a domain-specific application library parameterized by the spatial dimension, but in practice, this is hard to do: some pieces typically must be specialized to each dimensionality, and (in contrast to C++), Titanium does not provide a way to define template specializations, in which selected instantiations of a template are defined "by hand," while the template definition itself serves as a default definition when no specialization applies.

No type parameterization for methods. Java and C++ allow the programmer to supply type parameters on individual methods, as in the declarations

```
static <T> void fill(List<? super T> list, T obj) ...
static <T> List<T> emptyList () ...
```

from the Java library. For these examples, Titanium would have to make `T` a parameter of the containing class, which could cause notational inconvenience. In our particular collection of applications, we happen not to have had a pressing need for such definitions, but for a wider audience, they would probably have to be added to the language.

No type bounds. Like C++, but unlike Java, essentially any type parameter may be instantiated with any type as long as the class resulting from the substitution is semantically legal. In contrast, Java requires type bounds—in effect implicit specifications of the minimal contracts required of each type parameter. The advantage of Java's approach is that errors in a template instantiation may be explained solely by reference to the specification of the type parameters, without reference to the details of the body of the template, making the diagnosis of errors straightforward. In contrast, Titanium's approach amounts to a kind of syntactic macro (although the rules for what global names refer to what definitions are not quite that simple). The result is that as a practical matter programmers (but not formal semanticists) find Titanium's templates to be conceptually simpler. In any case, our experience has been that the potential programmer difficulties with diagnosing errors have not materialized.

Primitive types as parameters. In contrast to Java, Titanium allows primitive types as type parameters (as does C++). Java's chosen approach to generic programming causes all instantiations of a generic body to share all code and representation, making it impossible to use primitive types as parameters. Titanium uses a macro-expansion model, which requires no sharing of code between instantiations, and therefore no commonality of representation. This particular design choice seems essential to us. To get the effect of, for example, `List<double>` in Java, the programmer must substitute a *wrapper type* for `double`—a class whose instances are heap-allocated and each contain a `double` value (a practice typically known as *boxing* the primitive values). The same implementation considerations that apply to primitive types also apply to Titanium's immutable types (which include `Complex`), so that adopting the Java model would also require boxing these types (and thus largely defeating their purpose). The performance consequences of this extra level of indirection and of the required heap allocations are potentially enormous.

5 Application Experience

Since the purpose of Titanium is to support high performance scientific applications, we have found it essential from the beginning to create and evaluate application code written in Titanium. By using a combination of benchmarks, re-implementations of application codes developed in other languages, and new codes, we have been able to evaluate both the expressiveness of the Titanium language and the performance of its implementation. This continuing experience informs both improvements to the language design and improvements to the implementation technologies we create.

Our application experience includes the three NAS Benchmarks described in Section 4, along with the NAS Integer Sort (IS) and Embarrassingly Parallel (EP) kernels [4]. In addition, Yau developed a distributed matrix library that supports blocked-cyclic layouts and implemented Cannon's Matrix Multiplication algorithm, Cholesky and LU factorization (without pivoting). Balls and Colella built a 2D version of their Method of Local Corrections algorithm for solving the Poisson equation for constant coefficients over an infinite domain [5]. Bonachea, Chapman and Putnam built a Microarray Optimal Oligo Selection Engine for selecting optimal oligonucleotide sequences from an entire genome of simple organisms, to be used in microarray design. Our most ambitious efforts have been applications frameworks for Adaptive Mesh Refinement (AMR) algorithms and Immersed Boundary

(IB) Method simulations. In both cases these application efforts have taken a few years and were preceded by implementations of Titanium codes for specific problem instances, e.g., AMR Poisson [49], AMR gas dynamics [36] and IB for 1D immersed structures [39, 59].

5.1 Adaptive Mesh Refinment Framework

Since it was first developed by Berger and Oliger [8] for hyperbolic partial differential equations (PDEs), the AMR methodology has been successfully applied to numerical modeling of various physical problems. In Titanium, we have implemented a prototype of block-structured AMR following Chombo's software architecture [57]. Chombo [3] is a widely used AMR software package written in C++ and Fortran with MPI. Our Titanium implementation includes an infrastructure modeled after Chombo for supporting AMR applications and above it a solver for elliptic PDEs. We have provided the code to Cray and IBM as a benchmark to evaluate their new HPCS languages.

Almost all the Titanium features appear in our implementation of AMR. In particular, the set of grids at a particular level of refinement form a sparse coverage of the problem domain which requires the generality of Titaniums directory-based distributed arrays. The language support for domain calculus and subarrays facilitates the irregular computation of ghost values at various types of grid boundaries. Titanium templates provide very similar functionality to C++'s templates, allowing our infrastructure to mirror the templatized Chombo interface. Region-based memory management handles allocation and deallocation of temporary data structures. Overall, the Titanium implementation is more concise than its Chombo counterpart as shown in Section 5.3. Based on our case study [57], we find the performance of our AMR code matches that of Chombo on uniprocessors and SMP architectures. However, further work remains to match the scalability of Chombo on distributed-memory machines such as an SMP cluster – we believe the addition of message packing at some level is required to address the heterogeneity of communication cost.

5.2 Immersed Boundary Method

The immersed boundary method is a general approach for numerical modeling of systems involving fluid-structure interactions, where elastic (and possible active) tissue is immersed in a viscous, incompressible fluid. Peskin and McQueen first developed this method to study the patterns of blood flow in human hearts [37, 46]. It has subsequently been applied to a variety of problems, such as platelet aggregation during blood clotting, the swimming of eels, sperm and bacteria, and three-dimensional models of the cochlea. These and many other applications are described in a survey by Peskin [47].

We have developed a Titanium implementation of an immersed boundary method solver [26], and applied it to a variety of synthetic problems as well as modelling the cochlea [25] and the heart [58]. The complexity of the immersed boundary method comes from the interactions between the fluid and the boundaries of the material within it. The fluid is implemented as a uniform 3D mesh, while the materials consist of an arbitrary set of 1D, 2D, or 3D structures, which are not uniformly distributed and move during the simulation. The fluid and materials interact, which creates irregular patterns of communication between processors, and attempts to place material in the same memory partition as the nearby fluid creates load imbalance.

Titanium's object-oriented features and support for direct control over data layout facilitates the implementation of this class of numerical algorithms. The generic framework provides support for fluid flow simulation and fluid/material interaction, while each application of the method instantiates the generic material types with the unique features required by the physical entity. Our Titanium implementation uses a Navier-Stokes solver based on a 3D FFT, which calls FFTW using cross-language calls as described in Section 4.9. We implemented application-level message packing for better scalability on distributed-memory machines, suggesting, as for our AMR application, that some library or language support for such packing could be quite useful.

5.3 Expressiveness

Titanium does an excellent job handling subarrays, especially in hierarchical grid algorithms, which use them extensively. These algorithms usually leverage many of the other previously mentioned Titanium features as well, resulting in greater productivity and shorter, more readable code. As a rough comparison of language expressiveness, Figure 5 compares the line counts of Titanium with other implementations of the NAS MG benchmark and AMR.

In the case of MG, our Titanium implementation was compared with version 2.4 of the NAS MG code, written in Fortran with MPI. Note that only non-commented, timed code was included in the MG line counts. While the Titanium MG code is algorithmically similar to the NAS MG code, it is completely rewritten in the Titanium paradigm (i.e. one sided-communication in a Global Address Space memory model). The major difference between the Titanium and Fortran line counts is in communication – specifically, ghost cell updates. Titanium's domain calculus and array copy features concisely capture much of Multigrid's required functionality.

The line count comparison for AMR also overwhelmingly favors Titanium. The reasons are similar to those given for Multigrid. In general, the domain calculus functionality that had to be implemented as libraries in the C++/Fortran/MPI code is supported at the language level in Titanium. Note that both versions of the AMR code use templates, as discussed in Section 4.10, so this does not account for any difference in line counts.

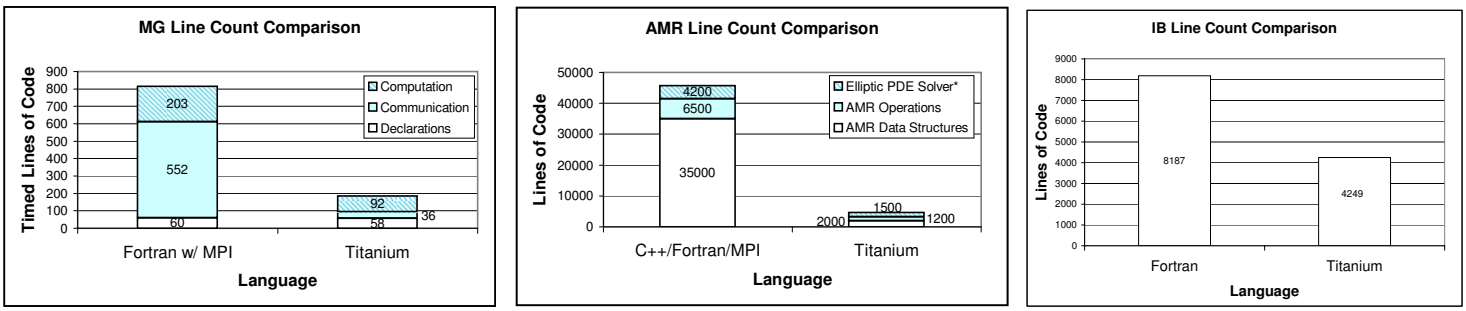

Figure 5: NAS MG, AMR and IB line count comparisons. Note that the C++ AMR Elliptic PDE Solver has more functionality than the Titanium version and that the Fortran IB version contains only vector annotations but no MPI code or other support for distributed memory parallelism.

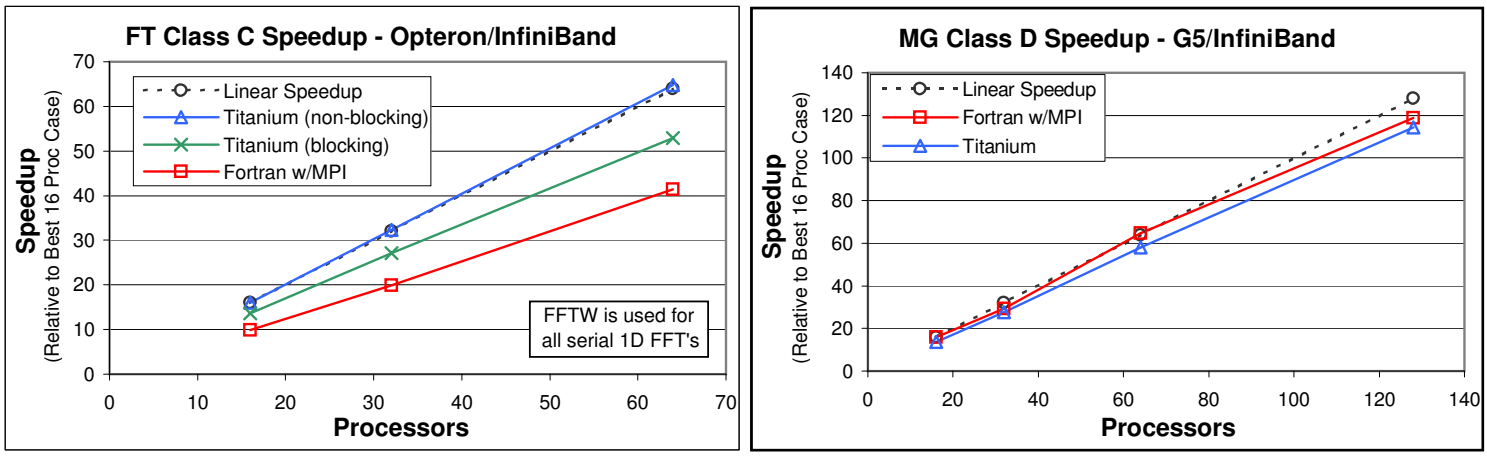

Figure 6: Speedup graphs for NAS FT and MG. The FT benchmark solves a 512^3 problem, while MG solves a 1024^3 problem. Note that all speedups are based on the best time (for either language) at 16 processors.

5.4 Performance

Despite Titanium's expressiveness, it is first and foremost a High-Performance Computing (HPC) language. In order to be attractive to programmers, an HPC language must demonstrate at least comparable performance to the parallel programming solutions in common usage, most notably Fortran+MPI.

Figure 6 compares Fortran and Titanium for the NAS FT and MG benchmarks running on two cluster systems detailed in Figure 7. The left graph dshows that the Titanium version of FT thoroughly outperforms Fortran, primarily because of 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 explains the performance gap between Fortran and the blocking Titanium code. Second, the best Titanium implementation also utilizes the non-blocking array copy feature, as explained in Section 4.5. 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 best Titanium code performs 36% faster than Fortran on 64 processors of the Opteron/InfiniBand system.

Note that both implementations of the FT benchmark use the same version of the FFTW library [22] for the local 1D 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.

The right side of Figure 6 provides a performance comparison for the NAS MG benchmark. Again, the Titanium version employs nonblocking array copy to overlap some of the communication time spent in updating ghost cells. However, the performance benefit is not nearly 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 demonstrate that Titanium performs only marginally worse than Fortran.

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

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
Alpha/ Elan3	Quad 1 GHz Alpha 21264 (750 nodes 4GB/node)	Quadrics QsNet1 Elan3 w/ dual rail (one rail used)	Tru64 v5.1, Elan3 libelan 1.4.20, Compaq C V6.5-303, HP Fortran Compiler X5.5A-4085-48E1K	PSC/Lemieux
Itanium2/ Elan4	Quad 1.4 Ghz Itanium2 (1024 nodes 8GB/node)	Quadrics QsNet2 Elan4	Linux 2.4.21-chaos, Elan4 libelan 1.8.14, Intel ifort 8.1.025, icc 8.1.029	LLNL/Thunder
x86/ Myrinet	Dual 3.0 Ghz Pentium 4 Xeon (64 nodes 3GB/node)	Myricom Myrinet 2000 M3S-PCI64B	Linux 2.6.13, GM 2.0.19, Intel ifort 8.1-20050207Z, icc 8.1-20050207Z	UC Berkeley/Millenium
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

Figure 7: Platforms on which Titanium was measured

6 Compilation and Runtime Technology

6.1 Source-to-source

Figure 8 illustrates the high-level system architecture of the Titanium implementation. The compiler translates Titanium code into C code, and then hands that code off to a C compiler to be compiled and linked with the Titanium runtime system and, in the case of distributed-memory back ends, with the GASNet communication system. We chose C as a target in order to achieve portability. Unfortunately, different machine architectures offer different sets of supported C compilers, which differ in optimization aggressiveness. On the same architecture, the performance difference from using one C compiler versus another on the generated C code can be as much as a factor of three. The one aspect the C compilers all have in common is that none were designed for compiling automatically generated C code. We sometimes find undiscovered bugs in the C compiler while compiling large Titanium programs.

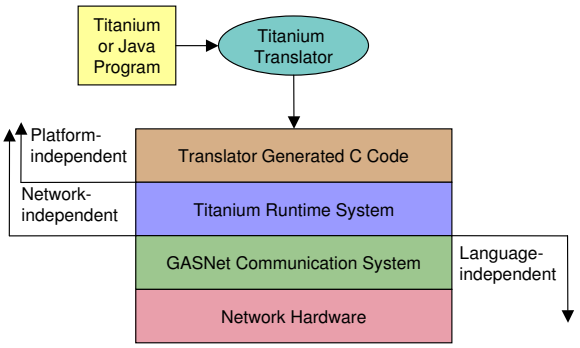

Figure 8: High-level system architecture of the Titanium implementation

The most challenging part of the interaction with the C compilers is to get the best serial performance out of the executable. Some C compilers' optimizations are very sensitive to the way the C code is written. To find such optimization opportunities, it is often necessary for the Titanium translator writer to examine the assembly code to check which optimizations were not applied.

An example is the code generation induced by the strides in Titanium arrays. Each Titanium array is allocated with a rectangular index domain. The *logical stride* of a domain specifies the difference between the coordinates of nearest neighboring points. A *physical stride,* a run-time quantity, is the difference between the addresses in memory of successive elements along one dimension. Strides are currently stored in variables in the generated C code. For array accesses in loops, we found that several C compilers were not able to unroll the loop generated by the Titanium compiler, but that we could enable the unrolling optimization by generating code with compile-time constant physical strides. Consequently, we are developing a stride-inference analysis for the Titanium compiler to statically determine strides with known constant values.

6.2 Program Analysis of Parallel Code

Aggressive program analysis is crucial for effective optimization of parallel code. Our analyses allow the Titanium front end to remove unnecessary operations and to provide more information to the back-end C compiler for use in its optimizations. Since Titanium is a dialect of Java, many Java analyses can be adapted to analyze Titanium programs. In addition, Titanium's structured parallelism simplifies program analysis. In this section, we describe some of the analyses used in the Titanium

translator.

6.2.1 Concurrency Analysis

Information about what sections of code may operate concurrently is useful for many optimizations and program analyses. In combination with alias analysis, for example, it allows the detection of potentially erroneous race conditions. We have used it to remove unnecessary synchronization operations and provide stronger memory consistency guarantees [30].

Titanium's textually aligned barriers and single-valued expressions place two important constraints on parallel programs:

1. The barriers in a Titanium program divide it into independent phases. Each textual instance of a barrier defines a phase, which includes all the expressions that can run after the barrier but before any other barrier. Since all threads must execute the same textual sequence of barriers and no thread may pass a barrier until all threads have reached it, all threads must be in the same phase. This implies that no two phases can run concurrently.

2. Titanium introduces the concept of single-qualification – the `single` type qualifier guarantees the qualified value is coherently replicated across all SPMD threads in the program, as explained in Section 2.3. Since a single-valued expression must have the same value on all threads, all threads must take the same branch of a conditional guarded by such an expression. If such a conditional is only executed at most once in each phase, then the different branches cannot run concurrently.

These two constraints allow a simple graph encoding of the concurrency in a program based on its control-flow graph. We have developed quadratic-time algorithms that can be applied to the graph in order to determine all pairs of expressions that can run concurrently [31].

6.2.2 Alias Analysis

Alias analysis identifies pointer variables that may, must, or cannot reference the same object. We use alias analysis to enable other analyses (such as concurrency analysis), and have considered also using it directly to find places where it is valid to introduce `restrict` qualifiers in the generated C code, enabling the C compiler to apply more aggressive optimizations. Applied to arrays, alias analysis can indicate Titanium arrays with identical physical strides or with compile-time constant strides, alleviating some of the problems in source-to-source translation described in Section 6.1.

The Titanium compiler's alias analysis is a Java derivative of Andersen's points-to analysis [2]. Each allocation site in a program is assigned an *abstract location*, and each variable has a set of abstract locations to which it can point. The implicit and explicit assignments in a program propagate abstract locations from source to destination. The analysis used in the Titanium compiler is *flow-insensitive*, and it iterates over all assignment expressions in a program in an unspecified order until a fixed point is reached. Two variables can then refer to the same memory location if their corresponding sets contain common abstract locations.

The analysis described thus far is purely sequential; it does not account for transmission of references, such as through a broadcast. The solution the Titanium compiler uses is to define two abstract locations for each allocation site: a local version that corresponds to an allocation on the current thread, and a remote version that corresponds to an allocation on some other thread. Transmission of an abstract location produces both the local and remote versions, since the source of the transmission isn't necessarily known at compilation time. Two variables on different threads can refer to the same location only if one variable can point to the remote version of an allocation site, and the other variable can point to either the local or the remote version of the same site.

The modified analysis is only a constant factor slower than the sequential analysis, and Titanium's SPMD model of parallelism implies that it only needs to be performed for a single thread. Its efficiency is thus independent of the number of runtime threads.

6.3 GASNet and One-Sided Communication

Titanium's global address space model relies on one-sided communication for performing remote reads and writes and encourages the use of remote accesses directly through application-level data structures; these yield smaller messages than when aggregation is done manually, as is common in message passing programming. Thus, a goal of our runtime work was to support fast one-sided communication that performs well even on small messages. While hardware limits absolute performance, our goal was to expose the best possible performance available on each network. Because the language and compiler technology have continued to evolve over the years, we also wanted the communication support to be extensible, so that runtime features like automatic packing [54] or software caching protocols could be implemented without redesigning the communication layer. Finally, we wanted a communication layer that could easily be ported across a variety of networks, since portability is essential to the success of any language. One portability strategy was to make the communication layer language-independent, which allowed us to leverage the implementation efforts of other active language efforts like UPC [55] and Co-Array Fortran [44].

The need for one-sided communication made traditional MPI [41] a poor target for our compiler, since emulation of one-side puts/gets on two-sided send/receive incurs unnacceptable performance penalties [15]. A careful analysis of MPI-2 one-sided

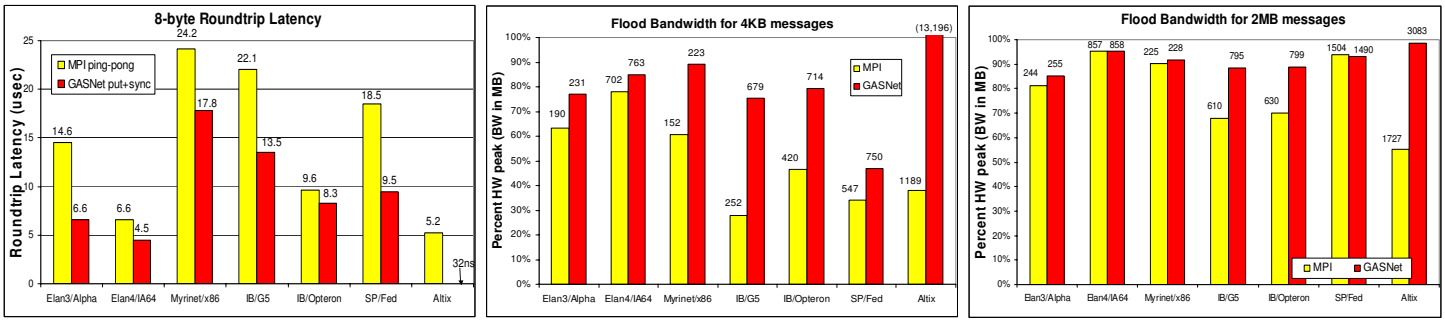

Figure 9: Microbenchmark performance comparison of GASNet and MPI across systems. The latency graph compares the *roundtrip* latency for an 8-byte raw MPI message-passing ping-pong (best case over MPI_Send/MPI_Recv or MPI_ISend/MPI_IRecv) against a GASNet blocking put operation (which blocks for the roundtrip acknowledgement). Bandwidth graphs show the peak bandwidth at 4KB and 2MB message sizes for a unidirectional message flood of the given size, and no unexpected MPI messages. Bandwidth bars are normalized to the hardware peak (a minimum of the I/O bus bandwidth and link speed), and labels show the absolute value in MB/s ($MB = 2^{20} bytes$).

also convinced us that it was designed for a bulk-synchronous style, which led to semantic restrictions that were unsuitable for arbitrary PGAS programs [40]. ARMCI offered many of our desired features, but did not support extensibility and enforced message ordering that we believed was an unnecessary performance handicap for a compilation target [43].

6.3.1 GASNet Overview

Titanium's distributed-memory backends are implemented using our GASNet communication system [14, 23], which provides portable, high-performance, one-sided communication operations tailored for implementing PGAS languages across a wide variety of network interconnects. The GASNet functionality is extensible using an Active Message abstraction [56], which can be used for remote locking, packing, and other features. In addition to Titanium, GASNet serves as the communication layer for two Unified Parallel C (UPC) compilers [9, 29], a Co-Array Fortran compiler [45], and some experimental projects. To date, the GASNet interface has been natively implemented on Myrinet (GM) [42], Quadrics QsNetI/QsNetII (elan3/4) [51], InfiniBand (Mellanox VAPI) [38], IBM SP Colony/Federation (LAPI) [33], Dolphin (SISCI) [21], Cray X1 (shmem) [7] and SGI Altix (shmem) [52]. These native implementations successfully expose the high-performance capabilities of the network hardware such as Remote Direct memory Acces (RDMA) to the PGAS language level, notably including automatic and efficient handling of issues such as memory registration on pinning-based networks using the novel Firehose [6] algorithm. Aside from the high-performance instantiations of the GASNet interface (conduits), there are also fully portable GASNet conduits for MPI 1.1 (for any MPI-enabled HPC system not natively supported), GASNet on UDP (for any TCP/IP network, e.g. Ethernet), and GASNet for shared-memory SMP's lacking interconnect hardware. Our GASNet implementation is written in standard C and is portable across architectures and operating systems – thus far it has been successfully used on over 16 different CPU architectures, 14 different operating systems, and 10 different C compilers, and porting existing GASNet conduits to new UNIX-like systems is nearly effortless.

GASNet's point-to-point communication API includes blocking and non-blocking gets/puts. Recent additions include support for non-contiguous data transfers and collective communication, both designed specifically for PGAS languages. The GASNet implementation is designed in layers for portability: a core set of Active Message functions constitute the basis for portability and extensibility, and we provide a reference implementation of the full API in terms of this core. In addition, the implementation for a given network can be tuned by implementing any appropriate subset of the general functionality directly upon the hardware-specific primitives.

6.3.2 GASNet Microbenchmark Performance

Figure 9 compares the latency and bandwidth microbenchmark performance of GASNet versus MPI across a range of systems, and shows that, contrary to popular belief, many networked cluster systems are better suited to the kind of one-sided communication found in Titanium than to MPI's two-sided model. All numbers were collected on large production supercomputers with multiple layers of switching, as detailed in Figure 7. Our data differs (especially in latency measurements) from many vendor-reported numbers, which are often reported on systems with zero or one levels of switching. The MPI data reported here is the best result obtained by running all available MPI implementations and running two different MPI benchmarks, one using blocking communication (from OSU [35]) and another using non-blocking communication.

The performance results demonstrate that GASNet matches or exceeds the performance of MPI in all cases, notably providing a significant improvement in small message roundtrip latencies and medium-sized message bandwidths. The primary explanation

for the performance gap is not related to clever implementation, but rather is semantic and fundamental: GASNet's put/get primitives were specifically designed to map very closely to the RDMA and distributed shared memory capabilities of modern interconnects – directly exposing the raw hardware performance while avoiding the well-known complexities and costs associated with message-passing (which include in-order delivery, message envelope matching, eager buffering copy costs, rendezvous messages, and unexpected message queueing costs). The differences are most pronounced on systems with good hardware support for remote memory access – the SGI Altix is an extreme example where the DSM hardware allows GASNet put/gets to expand to simple zero-copy load/store instructions (whose performance is limited only by cache performance), whereas the MPI implementation pays significant messaging overheads and extra copy costs.

7 Conclusions

The two main practical obstacles to high-performance scientific programming are the increasing complexity of its applications and the diversity, both in specification and performance characteristics, of the architectures that support it. Titanium attempts to address both—we believe successfully. By starting from an object-oriented base language, Java, Titanium allows programmers to exploit modern program-structuring techniques. Our choice of a global address space smooths the transition from familiar sequential programming to distributed programming by hiding many messy details of communication. Similarly, the implicit SPMD structure of the language hides messy details of parallel structure. At the same time, the partitioning of the address space gives programmers control over data layout, which is essential to performance and well-matched to the SPMD computation. Titanium's extensions to arrays directly address the management of many data-structuring details for a large class of scientific simulations.

At the same time, our experiences with a GASnet indicate that it is possible to support a programmer-friendly global memory model over a range of parallel architectures without undue sacrifice of performance. The use of a source-to-source translator ensures the necessary portability, and provides performance comparable to that of native compilers for languages with significantly less abstraction. The modifications we needed to the Java language itself to enhance performance were not extensive: immutable types, a template model allowing primitive type arguments, multi-dimensional arrays, and a region-based memory allocation feature.

Our application studies reveal the enormous potential in programmer productivity from a language like Titanium. The Titanium Immersed Boundary Method framework is the only implementation of this framework that runs on distributed memory parallel machines, in spite of over 30 years of ongoing development with the method and attempts to implement it in MPI. The Titanium implementation of the Chombo AMR framework has the same functionality as the Chombo code in MPI, but is significantly simpler and less than one tenth the size, in large part due to the global address space model and to built-in array types, which move software complexity into the language runtime where it can be re-used across application domains.

References

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

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] Applied Numerical Algorithms Group (ANAG). Chombo. http://seesar.lbl.gov/ANAG/software.html.

[4] 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.

[5] 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.

[6] C. Bell and D. Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *Workshop Communication Architecture for Clusters (CAC03) of IPDPS'03, Nice, France*, 2002.

[7] 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.

[8] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484–512, 1984.

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

[10] G. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie-Mellon University, September 1995.

[11] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[12] H. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[13] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

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

[15] D. Bonachea and J. C. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03)*, 2003.

[16] W. W. Carlson and J. M. Draper. Distributed data access in AC. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–47, Santa Barbara, California, United States, 1995.

[17] 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.

[18] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing (SC1993)*, 1993.

[19] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *Submitted*, 2005.

[20] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, August 1993.

[21] Dolphin Interconnect Solutions. *SISCI API User Guide, v1.0*, 2001. http://www.dolphinics.com.

[22] 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".

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

[24] D. Gay and A. Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[25] E. Givelberg and J. Bunn. A Comprehensive Three-Dimensional Model of the Cochlea. *To appear in The Journal of Computational Physics*, 2005.

[26] E. Givelberg and K. Yelick. Distributed Immersed Boundary Simulation in Titanium. Available from http://titanium.cs.berkeley.edu, 2003.

[27] High Performance Fortran Forum. High Performance Fortran Language Specification. http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20, Jan. 1997.

[28] IEEE and The Open Group. *Portable Operating System Interface (POSIX)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2004.

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

[30] 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.

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

[32] C. Kesselman. High performance parallel and distributed computation in compositional CC++. *ACM SIGAPP Applied Computing Review*, 4:24–26, Spring 1996.

[33] LAPI programming guide, 2003. IBM Technical report SA22-7936-00.

[34] 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.

[35] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *Int'l Journal of Parallel Programming*, 2004.

[36] 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.

[37] D. McQueen and C. Peskin. Computer-Assisted Design of Pivoting-Disc Prosthetic Mistral Valves. *Journal of Thoracic and Cardiovascular Surgery*, 86:126–135, 1983.

[38] Mellanox Technologies Inc. *Mellanox IB-Verbs API (VAPI)*, 2001. http://www.mellanox.com.

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

[40] MPI Forum. MPI–2: a message-passing interface standard. *International Journal of High Performance Computing Applications*, 12:1–299, 1998. http://www.mpi-forum.org/docs/mpi-20.ps.

[41] 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.

[42] Myricom. *The GM Message Passing System*. Myricom, Inc, GM v1.5 edition, July 2002.

[43] 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.

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

[45] Open64 project at Rice University. http://www.hipersoft.rice.edu/open64/.

[46] C. Peskin. *Flow Patterns Around Heart Valves: A Digital Computer Method for Solving the Equations of Motion*. PhD thesis, Albert Einstein College of Medicine, 1972.

[47] C. Peskin. The immersed boundary method. *Acta Numerica*, 11:479–512, 2002.

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

[49] 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.

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

[51] Quadrics Supercomputing. *Elan Programmer's Manual*.

[52] Man page collections: Shared memory access (SHMEM). http://www.cray.com/craydoc/20/manuals/S-2383-22/S-2383-22-manual.pdf.

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

[54] 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.

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

[56] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

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

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

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

[60] 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. *Concurrency: Practice and Experience*, 10:825–836, 1998.