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]

June 16, 2006

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 Partitioned Global Address Space model, SPMD parallelism support, multi-dimensional arrays and array-index calculus, memory management, immutable classes (class-like types that are value types rather than reference types), operator overloading, and generic programming. We provide an overview of the Titanium compiler implementation, covering various parallel analyses and optimizations, Titanium runtime technology and the GASNet network communication layer. We summarize results and lessons learned from implementing the NAS parallel benchmarks, elliptic and hyperbolic solvers using Adaptive Mesh Refinement, and several applications of the Immersed Boundary method.

1 Introduction

Titanium is an explicitly parallel dialect of Java™ designed for high-performance scientific programming [68]. 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 a wide range of parallel platforms that strikes an appropriate balance between expressiveness, user-provided information about concurrency and memory locality, 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 [20], CC++ [37], and AC [17], 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, which 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

*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.

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 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 experiences using it for scientific applications, and compilation and runtime innovations that support efficient execution on sequential and parallel platforms.

2 Serial Extensions to Java

We added several features to Java to better support scientific computation and high single-processor performance. In this section we illustrate these features, drawing on examples taken from our implementations of the three NAS Parallel Benchmarks [4, 21]: Conjugate Gradient (CG), 3D Fast Fourier Transform (FT), and Multigrid (MG). These benchmarks, like most scientific applications, rely heavily on multi-dimensional 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 language generality was motivated by more complicated parallel computations, such as Adaptive Mesh Refinement [66] and Immersed Boundary method simulation [55], which are more extensive applications that are described in section 6.

2.1 Titanium Arrays

In Java, all arrays inherit from `Object` and only 1D arrays are fully supported. All arrays have a starting index of zero, and there is no support for sub-arrays to share state with larger arrays. Multi-dimensional arrays in Java 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 multi-dimensionality.

Titanium extends Java with a powerful multi-dimensional array abstraction, which provides the same kinds of sub-array operations available in Fortran 90. Titanium arrays are indexed by integer tuples known as *points* and built on sets of points, called *domains*. The design is taken from that of FIDIL [31]. 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 smallest standard input (class A) to the NAS MG benchmark requires a 256^3 grid. The problem has periodic boundaries, which are implemented using 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 also has a type `Domain` that represents an arbitrary set of points, but Titanium arrays can only be built over `RectDomains` (i.e., rectangular sets of points). Titanium arrays 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, depending 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. On machines with hierarchical memory systems, `gridA` resides in memory with affinity to exactly one process, namely the process that executes the above statement. Similarly, objects reside in a single logical memory space for their entire lifetime (there is no transparent migration of data), however they are accessible from any process in the parallel program, as will be described in section 3.2.

2.2 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 many scientific codes, it is especially valuable in hierarchical grid algorithms like Multigrid and Adaptive Mesh Refinement (AMR). In a Multigrid computation on a regular mesh, there is a set of grids at various levels of refinement, and the primary computations involve sweeping over a given level of the mesh performing nearest neighbor computations (called stencils) on each point. To simplify programming, it is common to separate the interior computation from computation at the boundary of the mesh, whether those boundaries come from partitioning the mesh for parallelism or from special cases used at the physical edges of the computational domain. Since these algorithms typically deal with many kinds of boundary operations, the ability to name and operate on sub-arrays is useful. Java does not handle such

applications well, due to its non-contiguous memory layout and lack of sub-array support. Even C and C++ do not support sub-arrays well, and hand-coding in a 1D array can often confuse the compiler.

Titanium's domain calculus operators support sub-arrays 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, the following Titanium code creates two blocks that are logically adjacent, with a boundary of ghost cells around each to hold values from the adjacent block. The `shrink` operation creates a view of `gridA` by shrinking its domain on all sides, but does not copy any of its elements. Thus, `gridAInterior` will have indices from `[0,0,0]` to `[255,255,255]` and will share corresponding elements with `gridA`. The `copy` operation in the last line updates one plane of the ghost region in `gridB` by copying only those elements in the intersection of the two arrays. Operations on Titanium arrays such as `copy` are not opaque method calls to the Titanium compiler. The compiler recognizes and treats such operations specially, and thus can apply optimizations to them, such as turning blocking operations into non-blocking ones.

```
double [3d] gridA = new double [[-1,-1,-1]:[256,256,256]];
double [3d] gridB = new double [[-1,-1,256]:[256,256,512]];

// define interior for use stencil code
double [3d] gridAInterior = gridA.shrink(1);

// update overlapping ghost cells from neighboring block.
// gridB is the destination array, and gridAInterior is the source array
gridB.copy(gridAInterior);
```

The above example appears in the NAS MG implementation in Titanium [21], except that `gridA` and `gridB` are themselves elements of a higher level array structure. The `copy` operation as it appears here performs contiguous or discontiguous memory copies, and may perform interprocessor communication when the two grids reside in different processor memory spaces (see section 4.2). The use of a global index space across distinct array objects (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.

To implement periodic boundaries, one views an array as having been shifted in space, e.g., a block at the left-most end will be viewed as adjacent to the right-most. Titanium provides the `translate` operation for such index space shifts.

```
// update neighbor's overlapping ghost cells across periodic boundary
// by logically shifting the gridA to across the domain of gridB
gridB.copy(gridAInterior.translate([0,0,256]));
```

The `translate` method shifts the indices of the array view by logically adding the given point to every index in the array, creating a new view of `gridAInterior` where the relevant points overlap their boundary cells in `gridB`. The `translate` operation involves only construction of new array metadata (no data element movement), while the explicit `copy` operation performs the more expensive element copies. This separation helps to make the performance of the code transparent to programmers.

The ability to specify points as named constants can be used to write stencil operations such as those found in the NAS MG benchmark. The following code applies a 5-point 2D stencil to each point `p` in `gridAInterior`'s domain, where `gridAInterior` denotes the interior (non-boundary) portion of a grid for which the neighboring points are all defined. The results are written into another grid, `gridANew`, whose domain contains the same set of points as `gridA`. `S0` and `S1` are scalar constants determined by the specific stencil operator.

```
final Point<2> EAST =  [ 1, 0];
final Point<2> WEST =  [-1, 0];
final Point<2> NORTH = [ 0, 1];
final Point<2> SOUTH = [ 0,-1];
double [3d] gridANew = new double [gridA.domain()];

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

The full NAS MG code used for benchmarking in section 6.4 includes a 27-point stencil applied to 3D arrays. The Titanium code, like the NAS Fortran version of this benchmark, uses a manually-applied stencil optimization that eliminates redundant common subexpressions [18]. The `foreach` construct is explained in the next section.

2.3 Foreach Loops

Titanium provides an unordered looping construct, *foreach*, specifically designed for iterating through a multi-dimensional 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 gridAInterior.domain()) {
   gridB[p] = applyStencil(gridAInterior, 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.

This one loop concisely expresses an iteration over a multi-dimensional 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 multi-dimensional traversal. The *foreach* loop is a purely serial iteration construct—it is not a data-parallel construct.

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 [56, 58]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

3 Models of Parallelism

Designing parallelism facilities for a programming language involves a number of related decisions:

- Is parallelism expressed explicitly or implicitly?
- Is the degree of parallelism static or dynamic?
- How do the individual processes interact—how do they communicate data and synchronize with each other?

Answers to the first two questions have tended to group languages into principal categories: data-parallel, task-parallel, and Single Program Multiple Data (SPMD). Answers to the last question groups languages into message passing, shared memory, or Partitioned Global Address Space (PGAS). Here, we define these terms as used in this paper and explain the rationale behind our decision to use a SPMD control model and PGAS memory model in Titanium.

3.1 Creating Parallelism

Data Parallelism

Data-parallel languages like ZPL, NESL, HPF, HPJava, 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 [11, 12, 29, 40, 61]. These languages include array operators for element-wise 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 application 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 purely 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 parallelism, but complex dependence patterns such as those arising in parallel discrete-event simulation or sparse matrix factorization algorithms 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, and HPJava includes a library for making MPI calls. This reflects the tension between the elegance of pure data-parallelism and the application needs for more generality.

The second challenge for data-parallel languages is that the logical level of parallelism in the application is likely many times larger than the physical parallelism available on the machine. This is an advantage for users, since they need only express the parallelism that is natural in their problem, but it places an enormous burden on compiler and runtime system to handle resource

management. 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 has proven to be quite challenging for complex data layouts or for the case when multiple arrays with different distributions are involved.

Task Parallelism

At the opposite extreme from data-parallel languages are task-parallel languages, which allow users to dynamically create parallelism for arbitrary computations. Task-parallel systems include the Java thread model as well as languages extended with OpenMP annotation or threading libraries such as POSIX threads [32, 53]. *Parallel object-oriented languages* such as Charm++ and CC++ have a form of task parallelism in which method invocation logically results in the creation of a separate process to run the method body [34, 37]. These models allow programmers to express parallelism between arbitrary sequential processes[1], so they can be used for the most complicated sorts of parallel dependence patterns, but 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 processes to processors.

Static SPMD Parallelism

The Single Program Multiple Data (SPMD) model is a static parallelism model (popularized by systems such as MPI [48] and SHMEM [60]) in which a single program executes in each of a fixed number of processes that are created at program startup and remain throughout the execution. The parallelism is *explicit* in the parallel system semantics, in the sense that a serial, deterministic abstract machine cannot describe all possible behaviors in any straightforward way. 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.

The processes in an SPMD program synchronize with each other only at points chosen by the programmer, and otherwise proceed independently. Locking primitives or synchronous messages can be used to restrict execution order, and the most common synchronization construct in SPMD programs is the barrier, which forces all of the processes to wait for one another.

In the Titanium design, we chose the SPMD model to place the burden of parallel decomposition explicitly on the programmer, rather than the implementation, striving for a language that could support the most challenging parallel problems and give programmers a transparent model of how the computations would perform on a parallel machine. Our goal was to allow the expression of the most highly-optimized parallel algorithms.

3.2 Models of Sharing and Communication

The two basic mechanisms for communicating between processes are accessing shared variables and sending messages. Shared memory is generally considered easier to program, because communication is one-sided: processes can access shared data at any time without interrupting other processes, and shared data structures can be directly represented in memory. Message passing is more cumbersome, requiring both a two-sided protocol and packing/unpacking for non-trivial data structures, but it is also more popular on large-scale machines because it makes data movement explicit on both sides of the communication. For example, the popular MPI library provides primitives to send and receive data, along with collective communication operations to perform broadcasts, reductions, and many other global operations [48]. Message passing couples communication with synchronization, since message receipt represents completion of a remote event as well as data transfer. Shared-memory programming requires separate synchronization constructs such as locks to control access to shared data.

While these sharing models are orthogonal to the models for creating parallelism, there are common pairings. Both data parallelism and dynamic task parallelism are typically associated with shared memory, while SPMD parallelism is most commonly associated with message passing. However, Titanium and several other languages such as Unified Parallel C (UPC), Co-Array Fortran, Split-C, and AC couple the SPMD parallelism model with a variation of shared memory called a *Partitioned Global Address Space (PGAS)* [17, 20, 51, 64].

The term "shared memory" normally refers to a uniform memory-access-time abstraction, which usually means that all data is locally cacheable, and therefore can generally be accessed efficiently after the first access. A Partitioned Global Address Space offers the same semantic model with a different performance model: the shared-memory space is logically partitioned and processes have fast access to memory within their own partition, and potentially slower access to memory residing in a remote partition. In most PGAS languages, memory is also partitioned orthogonally into private and shared memory, with stack variables residing in private memory, and most heap objects residing in the shared space. A process may access any variable

[1]The terminology used for the individual sequential computations varies, unfortunately. In this paper, we will use the term *process,* except in contexts (such as Java or POSIX libraries) where *thread* is the preferred terminology.

located in shared space, but has fast access to variables in its own partition. PGAS languages typically require the programmer to explicitly indicate the locality properties of all shared data structures – in the case of Titanium, all objects allocated by a given process will always reside entirely in its own partition of the memory space.

Figure 1 illustrates a distributed linked list of integers in which each process has one list cell and pointers[2] in private space to list cells. The partitioning of PGAS memory may be reflected (as in Titanium) by an explicit distinction between *local* and *global* pointers: a local pointer must refer to an object within the same partition, while a global pointer may refer to either a remote or local partition. As used in Figure 1, instances of `l` are local pointers, whereas `g` and `nxt` are global pointers that can cross partition boundaries. The motivation for this distinction is performance. Global pointers 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 network communication is required to satisfy the access.

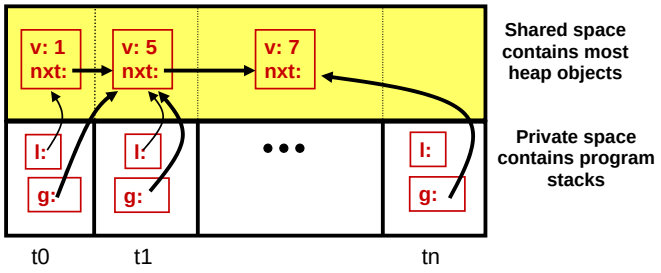

Figure 1: Titanium's Memory Model.

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. PGAS programs can run well on shared-memory multiprocessors and uniprocessors, where the partitioned-memory model need not correspond to any physical locality in hardware and the global pointers generally incur no overhead relative to local ones. Naively-written programs may ignore the partitioned-memory model and, for example, allocate all data structures in one process's shared-memory partition or perform fine-grained accesses on remote data. Such programs would run correctly on any platform but might deliver unacceptable performance on a distributed-memory platform where a higher cost is associated with access to data in remote partitions. 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 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 Parallel Extensions to Java

The standard Java language is ill-suited for use on distributed-memory machines because it adopts a dynamic task-parallel model and assumes a flat memory hierarchy. In this section, we describe the Titanium extensions designed to support efficient development and execution of parallel applications on distributed-memory architectures.

4.1 SPMD Parallelism in Titanium

Titanium's SPMD parallelism model is familiar to users of message-passing models such as MPI [48]. 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 processes and the index within that set of the executing process. The example prints these indices in arbitrary order. The number of Titanium processes is permitted to exceed 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 processes 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 are written in a bulk-synchronous style, Titanium also provides 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 processes 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:

[2]In this paper, we use the C/C++ term *pointer* to refer generically to values that may be dereferenced to yield objects. The term used in the Java specification is *reference*. Historically, the two terms were synonymous, and "reference" has its own meaning in C++.

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

Program statements executed between successive barriers are generally unconstrained, so that the use of textual barriers does not imply the kind of lock-step execution associated with data parallelism. Textual alignment of barriers enables the automated detection and prevention of program errors that can occur when one process skips a barrier unintentionally, leading to deadlocks or race conditions. It also turns out to be useful in certain program analyses, as described in section 7.2.

Single Qualification

The decision to require textual barrier alignment naturally led us to consider how to enforce this requirement: as a dynamic (run-time) check causing an exception when processes hit inconsistent barriers, or as a conservative static (compile-time) check. We decided early on to use static checks on the grounds that the category of errors associated with barrier-alignment violations could be rather obscure and in some cases (involving infinite loops) might not even be detectable by obvious dynamic checks. However, avoiding overly conservative static checks and unduly expensive analyses required that users provide some additional information.

Aiken and Gay developed the static analysis used by the Titanium compiler to enforce the barrier alignment restrictions, based on two key concepts [1]:

- A *statement with global effects* is one that must be textually aligned and thus invoked by all processes collectively. Such statements include those defined by the language to act as barriers, plus (conservatively) those that call methods that can execute statements with global effects (called *single methods*) and those that assign values to *single variables*—those with *single-qualified types,* defined below.
- A *single-valued expression* is roughly one whose successive evaluation yields the same sequence of values on all processes. Only single-valued expressions may be used in conditional expressions that affect which statements with global effects get executed.

As a result, all decisions on program paths leading to a barrier go the same way on all processes; each process executes the same sequence of barriers since it takes the same sequence of branches at critical points.

The only input required from the programmer to enforce the barrier-alignment rules is some explicit qualification of certain variables (local variables, instance variables, or parameters) and method return types as being single-valued. For this purpose, Titanium extends the Java type system with the `single` qualifier. Variables of single-qualified type may only be assigned values from single-valued expressions (similarly for method returns).

The rest of the analysis required to determine that programs satisfy the barrier alignment requirement is automatic. Determining that an expression is single-valued is a straightforward application of a recursive definition; single-valued expressions are defined to consist of literals, single-valued variables, calls to single-valued methods, and certain operators. The compiler determines which methods have global effects by finding barriers, assignments to single variables, or (transitively) calls to other single methods.

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 processes >
  Ti.barrier();
  < compute new local values >
  Ti.barrier();
}
```

We originally introduced single qualification to enable barrier-alignment analysis. We've since found that 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. However, our experience is that the use of single analysis can sometimes produce errors whose cause is obscure, as when the analysis detects that activating an exception on some processes might cause them to bypass a barrier or to fail to update a single-valued variable properly. Users seem to have mixed feelings: some find the static detection of problems to be useful and the need for single qualification to reflect natural notions about SPMD programming. On the other hand, as one might deduce from the space required for even an approximate and incomplete description, this area of the language has proven to be among the most subtle and difficult to learn.

4.2 Distributed Arrays

Titanium supports the construction of distributed array data structures in the Partitioned Global Address Space, in which each process creates its share of the total array. Since distributed data structures are explicitly built from local pieces rather than declared as distributed types, Titanium is sometimes referred to as a "local view" language. We have found that the generality of the pointer-based distribution mechanism combined with the use of arbitrary base indices for arrays provides an elegant and powerful mechanism for constructing shared data structures.

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 processor's 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 a temporary 1D array that is used to construct the "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`. Next, the `exchange` operation is used to create a replicated, global directory of pointers to the `myBlock` arrays on each process, which in effect makes `blocks` a distributed data structure. As shown in Figure 2, the `exchange` operation performs an all-to-all broadcast of pointers, and stores pointers to each processor's contribution in the corresponding elements 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. Accesses to points in the grid can then use a conventional 3D syntax. Both the block and cell positions are in global coordinates.

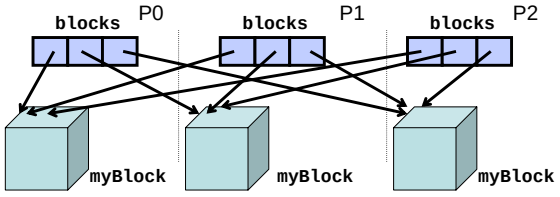

Figure 2: Distributed data structure created by Titanium's `exchange` operation for three processors.

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 6.1—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 demonstrated in section 2.2.

4.3 The Local Keyword and Locality Qualification

As illustrated in section 3.2, Titanium statically makes an explicit distinction between *local* and *global* pointers that reflects its PGAS memory model. A local pointer must refer to an object within the same process partition, while a global pointer may refer to an object in either a remote or local partition. Pointers in Titanium are global by default, but may be designated local using the `local` type qualifier.

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 that will be used for the local stencil computations and ghost cell surface updates.

Titanium's Partitioned Global Address Space model allows for fine-grained implicit access to remote data, but well-tuned Titanium applications perform most of their critical path computation on data that is either local or has been prefetched into local memory. This avoids fine-grained communication costs that 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 annotate the pointer to this process'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 pointer to a *local* pointer, 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). Adding the local qualifier to a pointer does not affect the distribution of the referenced data; it merely exposes the distribution properties explicitly for static analysis and documentation purposes. 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 that statically disables all the type and bounds checks required by Java semantics to save some computational overhead in production runs of debugged code.

The distinction between local and global pointers is modeled after Split-C, but Split-C pointers are local by default, whereas Titanium pointers are global by default. The global default makes it easier to port shared-memory Java code into Titanium, since only the parallel process creation needs to be replaced to get a functional parallel Titanium program. However, as noted in section 3.2, access to global pointers can be less efficient than local pointers. As will be shown in section 7.2, program analysis can be leveraged to automatically convert global to local pointers. Split-C's local default discourages the use of gratuitous global pointers, making such analyses less important in that language.

4.4 Non-blocking Array Copy

Although the array copy operation is conceptually simple, it can be expensive 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` Titanium array method to initiate non-blocking copying, and later ensure completion of the asynchronous communication using a second library call.

For example, Titanium's explicitly non-blocking array copy methods made it possible to considerably improve the speed of a 3D FFT solver. A straightforward implementation of this algorithm performs the FFT as two local 1D FFTs, followed by a 3D array transpose in which the processors collectively perform an all-to-all communication, followed by another local 1D FFT. This algorithm has two major performance flaws: processors sit mostly idle during the communication phase, and the intense communication during the transpose operation congests the interconnect and saturates at the bisection bandwidth of the network.

Both these issues can be dealt with using a slight reorganization of the 3D FFT algorithm employing non-blocking array copy. The new algorithm, which we have implemented in Titanium [21], first performs a local 1D FFT, followed by a local transpose and a second 1D FFT. However, unlike the previous algorithm, we begin sending each processor's portion of the grid (consisting of 2D planes) as soon as the corresponding rows are computed. By staggering the copies throughout the computation, the network is less likely to become congested and is more effectively utilized. Moreover, by using non-blocking array copy to send these slabs, we were able to hide nearly all of the communication latencies behind the local computation.

4.5 Regions

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 mechanisms with which to build memory allocators, and memory management issues occupy a significant portion of programmers' attention.

As a Java derivative, Titanium uses garbage collection (see section 7.3). However, implementing garbage collection for distributed programs with acceptable performance is still not entirely solved. We desired a mechanism that would give programmers some control over memory management costs as well as good locality properties 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 [26]. 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 abstract data structures involve hidden memory allocations. The built-in type `Domain` uses internal linked structures, for example. Consequently, 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.

5 Other Changes to Java

We have discussed the major departures of Titanium from Java that are directly applicable to parallel scientific computing. There are a number of other additions and variations that are of some interest to programmers, which we describe here.

5.1 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 (see the discussion of primitive types in section 5.3). Immutables provide efficient support for extending the language with new types that are manipulated and passed by value, avoiding pointer-chasing overheads that would otherwise be associated with the use of tiny objects in Java.

One compelling example of the use of immutables is for defining a complex number class, that is used to represent the complex values in the FT benchmark. In a straight Java version of such a class, each complex number is represented by an object with two fields, corresponding to the real and imaginary components, and methods that provide access to the components of 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—it imposes 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, a cumbersome method-call syntax would be required for performing operations on complex number objects in standard Java.

Titanium allows easy resolution of these performance issues by allowing the `immutable` keyword in class declarations. An immutable type is 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 implementation of `Complex` using immutables and operator overloading is available in the Titanium standard library and includes code like this:

```
public immutable class Complex {
  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. They are implicitly final, which means they never pay execution-time overheads for dynamic method call dispatch. All their instance variables are final, which makes their semantic distinction from ordinary classes less visible (as for standard Java wrapper classes such as `java.lang.Integer`). An array of `Complex` immutables is represented in memory as a single contiguous piece of storage consisting of all their real and imaginary components. This representation is significantly more compact in storage and efficient in runtime than objects 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,

the optional use of Titanium's `inline` method modifier provides a hint to the optimizer that calls to the given method should be inlined into the caller (analogous to the C++ inline modifier).

5.2 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 [39].

As further explained in section 7.1, the Titanium compiler is implemented as a source-to-source translator to C. This means that any library offering a C-compatible interface is potentially callable from Titanium (this also includes many libraries written in other languages such as C++ or Fortran). Since Titanium has no JVM, there is no need for a complicated calling convention (such as the Java JNI interface) to preserve memory safety.[3] 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 [24] to perform the local 1D FFT computations, thereby leveraging its auto-tuning features and machine-specific optimizations. 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 multi-dimensional 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 "multi-dimensional" arrays as 1D arrays of pointers to 1D arrays implies discontiguity of the array data that would have significantly increased the computational costs and complexity associated with calling external multi-dimensional computational kernels like FFTW.

5.3 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 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 of 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 such as `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 to not have had a pressing need for such definitions, but for a wider audience, they would probably have to be added to the language.

[3] 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 that are typically used by standard Java implementations.

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.

5.4 Other Significant Deviations from Java

Titanium is mostly a superset of Java, and consequently most serial Java code can run unmodified as part of a Titanium application. We leverage this fact in our implementation of the Java standard libraries, which are reused almost unmodified from the standard Sun Java compiler distribution. However, Titanium imposes a few significant restrictions relative to standard Java that are worthy of mention.

As described in section 4, Titanium adopts the SPMD model of static parallelism, whereby the parallel processes are all implicitly created at program startup. Consequently, Titanium does not support the Java-style creation of new threads of execution at runtime, nor any libraries that rely on this feature for correct operation (notably including `java.awt` and `java.net`). We have investigated the possibility of allowing dynamic thread creation within a Titanium process; however this remains an open area of investigation.

Unlike most Java compilers, our Titanium compiler performs whole-program compilation (primarily to assist interprocedural compiler optimization), and the final compilation result is a native, static binary executable—Titanium has no Java Virtual Machine (JVM), no Just-in-time (JIT) optimizer, and no class files. As a consequence of these design decisions, Titanium does not support dynamic class loading (the Java feature that allows new sections of program code to be loaded at runtime). We've found this restriction to be acceptable for our targeted class of scientific applications, and the improvements it enables in compiler analysis and optimization make the tradeoff worthwhile.

Finally, Titanium defines a set of "Erroneous Exceptions," which are considered Titanium programming errors and thus not guaranteed to provide precise exception behavior as in standard Java. This category includes runtime exceptions such as `NullPointerException` and `IndexOutOfBoundsException` that are *implicitly* generated by program statements that attempt to deference a null pointer or access an array out of bounds. Such programming errors are still detected and reported by the Titanium runtime system; however, they are considered fatal errors, and subsequent program behavior is undefined. This restriction allows for more aggressive optimization of array accesses in loops, which is crucial for performance in scientific applications. This design decision is in agreement with general scientific programming practice, where fine-grained program errors are usually considered fatal and error recovery is handled (if at all) by using coarse-grained mechanisms such as checkpointing.

For further details on these and other minor deviations from standard Java, consult the Titanium language reference [30] and compiler release notes.

6 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 2, 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 application 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 [57], AMR gas dynamics [43] and IB for 1D immersed structures [46, 67].

6.1 Adaptive Mesh Refinement Framework

Since it was first developed by Berger and Oliger [9] 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 [66]. 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 an elliptic PDE solver that uses the framework. The code is being used by both Cray, Inc. and IBM as a benchmark to evaluate their new HPCS languages.

Almost all the Titanium features appear in our implementation of AMR. The AMR computation uses stencil operations as in NAS MG, but the data structure itself is a hierarchy of grids at various levels of refinement. The set of grids at a particular level of refinement forms a sparse coverage of the problem domain that requires the generality of Titanium's directory-based distributed arrays. Figure 3 diagrams a portion of a grid hierarchy in two dimensions; in a real application there are generally several levels of refinement, but only two are shown here. The stencil operations may require values from grids that are on neighboring grids at the same level or different levels. Each grid is owned by a particular process, and for load balancing reasons, the set of grids are distributed to roughly balance computational load. While some attention is paid to locality in the load balancing algorithms, a neighboring grid is likely to be owned by a different process. Titanium's language support for domain calculus and sub-arrays facilitates the irregular computation of ghost values at the various types of grid boundaries. The Titanium code below demonstrates the exchange of the intersection values between grids on the same level.

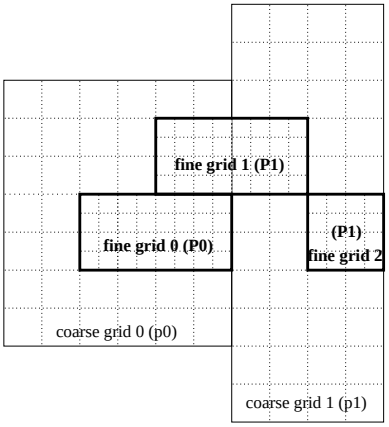

Figure 3: Two levels of an adaptive 2D mesh

```
for (int i = 0; i < numOfGrids; i++) {
    GridData curGrid = myGrids[i];
    GridData [1d] neighbors = curGrid.getNeighbors();
    for (int j = 0; j < curGrid.neighborCount; j++) {
        curGrid.copy(neighbors[j]);
    }
}
```

`GridData` is a user-defined wrapper class for a Titanium array. The `copy` method of the `GridData` class simply invokes the array copy method of the Titanium array it represents. The array copy method call computes the intersection domain of `curGrid` and its neighbor, and the values in the intersection are copied from the neighbor to `curGrid`. When the neighbor is remote, communication is performed by the runtime system. In AMR, the average intersection size is around 4K bytes, which implies that many of the data transfers are too small to achieve peak bandwidth on modern networks. The compiler performs automatic packing to aggregate the small messages using an SMP-aware communication model.

Titanium templates provide very similar functionality to C++ 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 6.3. Based on our case study [63], we find the performance of our AMR code matches that of Chombo on uniprocessors and SMP architectures. It also shows comparable scalability in comparison to Chombo on an IBM SP machine using up to 112 processors.

6.2 Immersed Boundary Method

The Immersed Boundary method is a general approach for numerical modeling of systems involving fluid-structure interactions, where elastic (and possibly 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 [44, 54]. 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

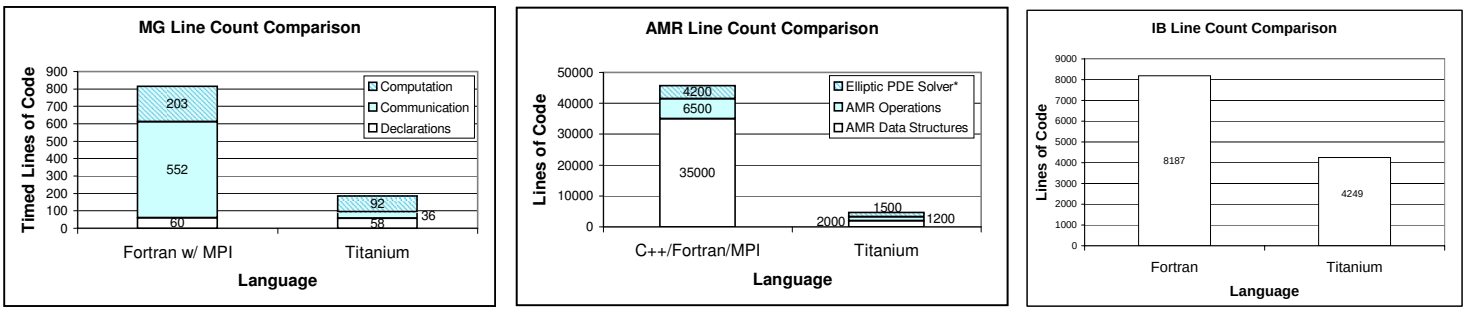

Figure 4: NAS MG, AMR and IB line count comparisons. The Elliptic PDE Solver module of the C++ AMR code has more functionality than the Titanium version and that the Fortran IB version contains only vector annotations but no MPI code or other support for distributed-memory parallelism.

models of the cochlea. These and many other applications are described in a survey by Peskin [55]. The applications are computationally intensive–for example, a single heartbeat required 100 CPU hours on a Cray vector machine. However, there were no distributed-memory implementations of the 3D method prior to our work [28, 46, 67].

We have developed a Titanium implementation of an Immersed Boundary method solver [28] and applied it to a variety of synthetic problems as well as modeling the cochlea [27] and the heart [44]. 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 non-uniformly distributed and move during the simulation. The fluid and materials interact, creating irregular patterns of communication between processors, and attempts to place material in the same memory partition as the nearby fluid result in 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 5.2. We implemented application-level message packing for better scalability on distributed-memory machines, and are investigating the use of library and language support for automated aggregation to provide competetive performance without the burden of manual packing.

6.3 Expressiveness

Titanium does an excellent job of handling sub-arrays, 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 4 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 reference implementation, written in Fortran with MPI. 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 Partitioned 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. Both versions of the AMR code use templates, as discussed in section 5.3, so this does not account for any difference in line counts.

6.4 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 5 compares Fortran and Titanium for the NAS FT and MG benchmarks running on the two InfiniBand cluster systems detailed in Figure 6. The left graph shows that the Titanium version of FT thoroughly outperforms the standard Fortran+MPI implementation, 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 explains the performance

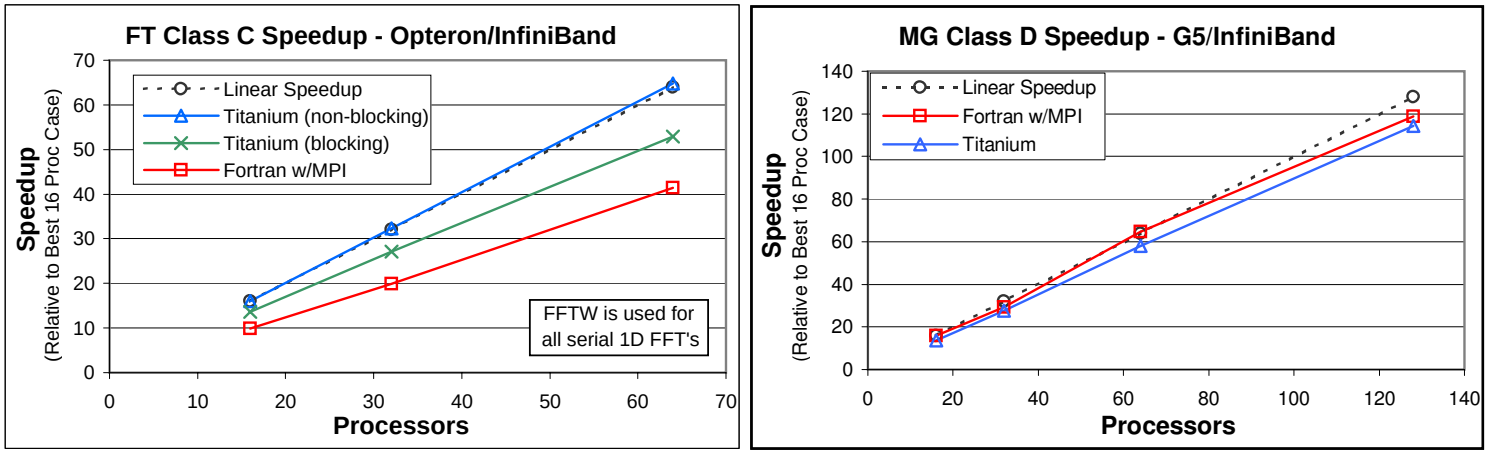

Figure 5: Speedup graphs for NAS FT and MG. The FT benchmark solves a 512^3 problem, while MG solves a 1024^3 problem. All speedups are based on the best time (for either language) at 16 processors.

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 4.4. 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 the reference Fortran code on 64 processors of the Opteron/InfiniBand system.

Both implementations of the FT benchmark use the same version of the FFTW library [24] 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. Bell, Bonachea, Nishtala and Yelick demonstrated that similar communication optimizations can be expressed in the Fortran+MPI code, but doing so delivers less impressive performance improvements due to the higher overheads associated with fine-grained MPI message passing relative to the lighter-weight one-sided communication operations exposed by PGAS languages [7].

The right side of Figure 5 provides a performance comparison for the NAS MG benchmark. Again, the Titanium version employs non-blocking 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. Nevertheless, the results demonstrate that Titanium performs only marginally worse than Fortran for MG. The MG performance scales fairly linearly, as expected since the domain decomposition is well load-balanced for all the measured processor counts.

The larger Titanium applications are still undergoing scalability analysis and tuning, and inherent challenges such as load imbalance and high communication to computation ratios make linear scaling unlikely in any language. As described in section 6.1, we find the performance of our AMR code closely matches that of C++/Fortran Chombo on uniprocessors and SMP architectures, and provides comparable scalability. For the Immersed Boundary method, there is no MPI equivalent against which to compare, but our code runs on distributed-memory machines, where it outperforms previous shared-memory implementations and enables new application simulations. The code demonstrates a 5x speedup on the 512^3 problem size in moving from 16 to 128 processors on an IBM SP [28]. We believe the use of non-blocking communication combined with hardware that supports communication overlap will benefit both codes.

7 Compilation and Runtime Technology

We have shown that Titanium's performance is comparable to that of other parallel programming environments. Achieving this level of performance and portability required extensive work both in the compiler and the runtime layer, though we leveraged existing solutions where possible.

7.1 Source-to-source Compilation

Figure 7 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 the vendor-provided C compiler to be compiled and linked with the Titanium runtime system and, in the case of distributed-memory backends, with the GASNet communication system. Titanium compilation incorporates uses of the Titanium standard library and optimizes instances of its use as appropriate. Unlike HPJava, we chose C as a compilation target instead of Java bytecode in order to maximize portability, since several high-end supercomputers such as the Cray X1 and the IBM Blue Gene lack Java compilers and possibly the OS support required for a fully-operational

System	Processor	Network	Software	Location
Alpha/ Elan3	Quad 1 GHz Alpha 21264 (750 nodes 4GB/node)	Quadrics QsNet1 Elan3 dual rail (one rail used)	Tru64 v5.1, Elan3 libelan 1.4.20, Compaq C V6.5-303, HP Fortran Compiler X5.5A-4085	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 (64 nodes 3GB/node)	Myricom Myrinet 2000 M3S-PCI64B	Linux 2.6.13, GM 2.0.19, Intel ifort 8.1.029, icc 8.1.033	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
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
SP/ Federation	8-way 1.5 GHz Power4 (272 nodes 32GB/node)	IBM Federation	IBM AIX 5.2, IBM LAPI v2.3.3.3, IBM XLC/XLF 6.0	SDSC/Datastar

Figure 6: Platforms on which Titanium was measured

JVM. 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 – consequently, we sometimes find undiscovered bugs in the C compilers while compiling large Titanium programs.

The most challenging part of the interaction with the C compilers is the effort required to achieve the best serial performance for the compiled 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.

One 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. The *physical stride,* a run-time quantity, is the difference between the addresses in memory of successive elements along one dimension. Strides are currently stored as variables in the generated C code. For array accesses in loops, we found that several C compilers were unable 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.

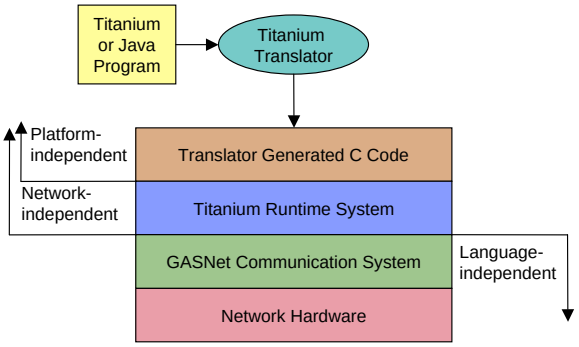

Figure 7: High-level system architecture of the Titanium implementation

7.2 Program Analysis of Parallel Code

Aggressive program analysis is crucial for effective optimization of parallel code. Our analyses allow the Titanium front end compiler 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 Titanium-specific analyses used in the Titanium optimizer.

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 [35].

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 processes must execute the same textual sequence of barriers and no process may pass a barrier until all processes have reached it, all processes 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 processes in the program, as explained in section 4.1. Since a single-valued expression must have the same value on all processes, all processes 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 [36].

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 race detection), 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 7.1.

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 pointers, 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 process, and a remote version that corresponds to an allocation on some other process. 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 processes 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 process. Its efficiency is thus independent of the number of runtime processes.

Local Qualification Inference

As described in section 4.3, pointers in Titanium can be declared as local or global. By default, pointers are global, since this places fewer restrictions on their use than does local qualification. This generality comes at a cost, however, since global pointers are less space- and time-efficient than local pointers. Manually inserting local qualifiers into user code can be tedious and error-prone, and is impossible in the case of the standard libraries.

The Titanium optimizer includes a *local qualification inference* (LQI) that automatically determines a conservative set of pointers that can be safely converted to local. Using a constraint-based inference, it automatically propagates locality information gleaned from allocation statements and programmer annotations through the application code [41]. Local qualification enables several important optimizations in the implementation of pointer representation, dereferencing, and array access. These optimizations reduce serial overheads associated with global pointers and enable more effective optimization and code-generation by the backend C compiler. Figure 8 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

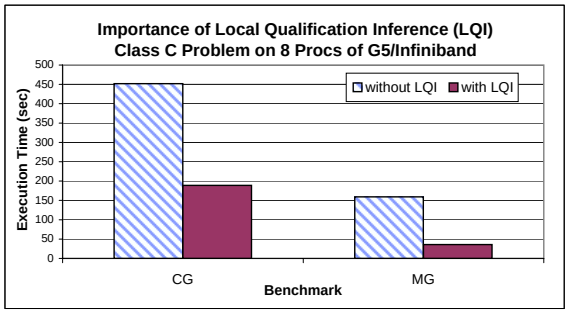

Figure 8: Performance speedup obtained with the LQI compiler optimization

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.

Converting Blocking to Non-Blocking Operations

An array copy is a bulk communication operation in Titanium between two arrays. The contents of the source array are copied to the destination array where the domains of the two arrays intersect. This operation is semantically blocking, which means that the process executing the array copy waits until the operation completes before moving on to the next instruction. If either the source array or the destination array is remote, then communication over the network is required. Although a non-blocking array copy interface is exposed to the programmer, using non-blocking array copy explicitly in the application code can complicate the code and introduce synchronization errors that are difficult to find.

A compiler optimization that automatically converts blocking array copies into non-blocking operations has been developed in the Titanium compiler. The goal of this optimization is to push the synchronization that ensures the completion of the array copy as far down in the instruction stream as possible, allowing subsequent computation and communication operations to be overlapped with the communication latency of the array copy.

Inspector/Executor

The Titanium compiler has support for the *inspector/executor* framework [62] to optimize irregular remote accesses of the form `a[b[i]]` that appear in a loop. The array access pattern is computed in an initial *inspector* loop. All the required elements are prefetched into a local buffer. The *executor* loop then uses the prefetched elements for the actual computation. Different methods of communication can be used for prefetching data into a local buffer. These include:

1. Pack method: only communicates the needed elements without duplicates. The needed elements are packed into a buffer before sending them to the processor that needs the data.
2. Bound method: a bounding box that contains the needed elements is retrieved.
3. Bulk method: the entire array is retrieved.

The best communication method is both application and machine specific. The application determines the size of the array, number of accesses to that array, and the size of the bounding box. The machine gives different memory and communication costs. The Titanium compiler generates code that can choose the best communication method at runtime based on a performance model.

7.3 Memory Management

In order to minimize our implementation effort and to achieve portability, we chose to adopt the widely used Boehm-Demers-Weiser conservative garbage collector [13, 14] 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 [22]). 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. This was a principal motivation for introducing region-based memory management into the language (see section 4.5).

7.4 GASNet and One-Sided Communication

Titanium's Partitioned 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; this tends to 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 will of course limit absolute performance, our goal was to expose the best possible performance available on each network. Because language and compiler technology continue to evolve over the years, we also wanted the communication support to be extensible, so that runtime features such as automatic packing [62] 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 important strategic decision was to make the communication layer language-independent, allowing us to leverage this implementation work in the context of other active parallel language efforts such as UPC [64] and Co-Array Fortran [51].

The need for one-sided communication made traditional MPI [48] a poor target for our compiler, since emulation of one-sided put/get operations over two-sided message passing incurs unacceptable performance penalties [16]. A careful analysis of the MPI-2 one-sided interface [47] also convinced us that it was designed primarily for monolithic application development and imposes semantic restrictions that make it unsuitable as a compilation target for parallel languages. ARMCI [50] offers some of our desired features, but lacks extensibility and enforces message ordering that we believed was an unnecessary performance handicap for a compilation target.

GASNet Overview

Titanium's distributed-memory backends are implemented using our GASNet communication system [15, 25], which provides portable, high-performance, one-sided communication operations tailored for implementing PGAS languages across a wide variety of network interconnects. GASNet provides extensibility using an Active Message abstraction [65], which can be used to implement remote locking, distributed garbage collection, and other language-specific communication operations. In addition to Titanium, GASNet serves as the communication layer for two Unified Parallel C (UPC) compilers [10, 33], a Co-Array Fortran compiler [52], and some experimental projects. To date, the GASNet interface has been natively implemented on Myrinet (GM) [49], Quadrics QsNetI/QsNetII (elan3/4) [59], InfiniBand (Mellanox VAPI) [45], IBM SP Colony/Federation (LAPI) [38], Dolphin (SISCI) [23], Cray X1 (shmem) [8] and SGI Altix (shmem) [60]. These native implementations successfully expose the high-performance capabilities of the network hardware such as Remote Direct Memory Access (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 puts and gets that are fully one-sided and decoupled from inter-process synchronization, with no relative ordering constraints between outstanding operations. 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. Our research using GASNet has shown that the layered design approach is effective at providing robust portability as well as high performance up to the application level on architectures ranging from loosely coupled clusters with a near-commodity network [19] to tightly coupled MPP systems with a hardware-supported global memory system[8].

GASNet Microbenchmark Performance

Figure 9 compares the latency and bandwidth microbenchmark performance of Titanium with 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 message-passing model. All numbers were collected on large production supercomputers with multiple layers of switching, as detailed in Figure 6. Our data differs (especially in latency measurements) from many vendor-reported numbers, which are often collected under laboratory conditions 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 [42]) and another using non-blocking communication. In all cases the MPI performance shown notably does *not* include the cost of emulating the put/get operations required by PGAS languages over MPI message-passing, which would add considerable additional overhead [16].

The performance results demonstrate that Titanium's GASNet communication layer matches or exceeds the performance of MPI in all cases, notably providing a significant improvement in small message round-trip 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 queuing costs). The differences are even more pronounced on systems with good hardware support for remote memory access such as the SGI Altix or Cray X-1 [8]. On such systems, the

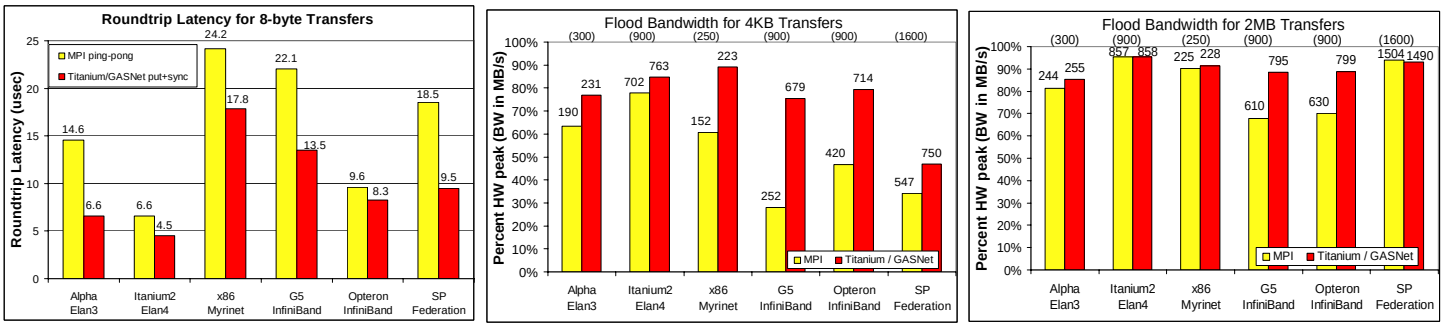

Figure 9: Microbenchmark performance comparison of Titanium/GASNet versus MPI across systems. The latency graph compares the *round-trip* latency for an 8-byte raw MPI message-passing ping-pong (best case over MPI_Send/MPI_Recv or MPI_ISend/MPI_IRecv) against a Titanium/GASNet blocking put operation (which blocks for the round-trip acknowledgment). Bandwidth graphs show the peak bandwidth at 4KB and 2MB transfer 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, as shown in parentheses), and labels show the absolute value in MB/s ($MB = 2^{20}$ *bytes*).

hardware allows GASNet put/gets to expand to simple zero-copy load/store instructions (whose performance is often limited only by memory system performance), whereas the MPI implementation pays significant messaging overheads and extra copy costs.

8 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. Titanium allows programmers to exploit modern program-structuring techniques by starting from an object-oriented base language (Java). Our choice of a global address space smoothes the transition from familiar sequential programming to distributed programming by hiding many of the low-level complexities of network communication. Similarly, the SPMD nature of the language simplifies communication and synchronization among processes while avoiding the limitations of data-parallel languages. At the same time, the partitioning of the shared address space gives programmers explicit control over data layout, which is essential to performance and well-matched to the SPMD computation model. Titanium's extensions to arrays concisely and efficiently address the management of many data-structuring details for a large class of scientific simulations.

At the same time, our experiences with 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 the included analyses and optimizations provide performance comparable to that of native compilers for languages with significantly less abstraction and productivity features. The modifications made to the Java language itself to enhance serial and parallel performance were not extensive: immutable types, a template model allowing primitive type arguments, multi-dimensional arrays, locality qualification, and a region-based memory allocation feature.

Our application studies reveal the enormous potential in programmer productivity from a language like Titanium, as evidenced by the significant decrease in code size compared to other languages. Features such as built-in domain types, templates, operator overloading, foreach loops, and support for cross-language development enhance productivity and promote code reuse. The Titanium Immersed Boundary method framework is the only implementation of this framework that runs on distributed-memory parallel machines, in spite of years of ongoing development with the method and its applications. The Titanium implementation of the Chombo AMR framework has the same functionality as the Chombo code in MPI, but is significantly cleaner and less than one tenth the size, in large part due to the global address space model and to the powerful array and domain-calculus abstractions, which move software complexity into the language runtime where it can be reused 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, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[8] 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.

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

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

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[22] 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.

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

[24] 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".

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

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

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

[28] E. Givelberg and K. Yelick. Distributed Immersed Boundary Simulation in Titanium. *To appear in SIAM Journal of Scientific Computing*, 2006.

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

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

[31] P. N. Hilfinger and P. Colella. FIDIL: A language for scientific programming. In R. Grossman, editor, *Symbolic Computing: Applications to Scientific Computing*, Frontiers in Applied Mathematics, chapter 5, pages 97–138. SIAM, 1989.

[32] 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.

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

[34] L. V. Kale, M. Hills, and C. Huang. An orchestration language for parallel objects. In *Proceeding of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 04)*, 2004.

[35] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, November 2005.

[36] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

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

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

[39] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[40] H.-K. Lee, B. Carpenter, G. Fox, and S. B. Lim. HPJava: Programming support for high-performance grid-enabled applications.

[41] 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.

[42] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int'l Journal of Parallel Programming*, 2004.

[43] 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.

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

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

[46] S. Merchant. Analysis of a contractile torus simulation in Titanium. Masters Report, Computer Science Division, University of California Berkeley, August 2003.

[47] 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.

[48] 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.

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

[50] 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.

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

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

[53] OpenMP specifications. http://www.openmp.org.

[54] 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.

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

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

[57] 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.

[58] G. R. Pike. *Reordering and Storage Optimizations for Scientific Programs*. PhD thesis, University of California, Berkeley, January 2002.

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

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

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

[62] 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.

[63] J. Z. Su, T. Wen, and K. A. Yelick. Compiler and runtime support for scaling adaptive mesh refinement computations in Titanium. Technical Report UCB/EECS-2006-87, EECS Department, University of California, Berkeley, June 13 2006.

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

[65] 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.

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

[67] S. M. Yau. Experiences in using Titanium for simulation of immersed boundary biological systems. Masters Report, Computer Science Division, University of California Berkeley, May 2002.

[68] 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.