Titanium: A High-Performance Java Dialect*

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto,
Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger,
Susan Graham, David Gay, Phil Colella, and Alex Aiken

Computer Science Division
University of California at Berkeley
and
Lawrence Berkeley National Laboratory

Abstract

Titanium is a language and system for high-performance parallel scientific computing. Titanium uses Java as its base, thereby leveraging the advantages of that language and allowing us to focus attention on parallel computing issues. The main additions to Java are immutable classes, multi-dimensional arrays, an explicitly parallel SPMD model of computation with a global address space, and zone-based memory management. We discuss these features and our design approach, and report progress on the development of Titanium, including our current driving application: a three-dimensional adaptive mesh refinement parallel Poisson solver.

1 Overview

The Titanium language is designed to support high-performance scientific applications. Historically, few languages that made such a claim have achieved a significant degree of serious use by scientific programmers. Among the reasons are the high learning curve for such languages, the dependence on "heroic" parallelizing compiler technology and the consequent absence of compilers and tools, and the incompatibilities with languages used for libraries. Our goal is to provide a language that gives its users access to modern program structuring through the use of object-oriented technology, that enables its users to write explicitly parallel code to exploit their understanding of the computation, and that has a compiler that uses optimizing compiler technology where it is reliable and gives predictable results. The starting design point for Titanium is Java. We have chosen Java for several reasons.

*This work was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense under contracts F30602-95-C-0136 and DABT63-96-C-0056, by the U.S. Department of Energy under contracts DE-FG03-94ER25206, DE-AC03-76SF00098 (through the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences), and W-7405-ENG-48 (through Lawrence Livermore National Laboratory, subcontract No. B336568), by the National Science Foundation, under contracts CDA-9401156, by the Army Research Office under contract DAAH04-96-1-0079, and by a Microsoft Graduate Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1. Java is a relatively small and clean object-oriented language (compared, for instance, to C++) and therefore is easy to extend.
2. Java is popular and is itself based on popular languages (C/C++). Learning Titanium requires little effort for those acquainted with Java-like languages.
3. Java is a safe language, and so is Titanium. Safety enables the user to write more robust programs and the compiler to perform better optimization.

Titanium's main goals are, in order of importance: performance, safety, and expressiveness.

- **Performance** is a fundamental requirement for computationally demanding scientific applications. Many design choices reflect this goal. For instance, the execution model is explicitly parallel, thus eliminating the need for a parallelizing compiler. Any distributed data structure is fully defined by the programmer, who has complete control over its layout across process boundaries. In addition, the programmer may use type modifiers in variable declarations to convey locality information that the compiler may find hard to infer reliably from static analysis.
- **Safety** has two meanings in Titanium. One is the ability to detect errors statically. For instance, the Titanium compiler can ensure that all processes will execute the correct sequence of global synchronizations. The other is the ability to detect and report run-time errors, such as out-of-bound indices, accurately. Both forms of safety facilitate program development; but, not less importantly, they enable more precise analysis and more effective optimizations.
- **Expressiveness.** Because of the priorities of our target customers, we sacrificed expressiveness to the first two goals as necessary. However, with built-in features such as true multi-dimensional arrays and iterators, points and index sets (including irregular ones) as first-class values, and references that span processor boundaries (similar to global pointers in Split-C), Titanium is far more expressive than most languages with comparable performance.

Titanium is based on a parallel SPMD (for Single Program, Multiple Data) model of computation. In this model, the parallelism comes from running the same program asynchronously on n processors, each processor having its own data. Titanium processes can transparently read and write data that resides on other processors. Titanium programs run on both distributed-memory and shared-memory architectures. (Support for SMPs is not based on Java threads; see Section **??**.) However, a Titanium program written for an SMP is not guaranteed to run efficiently on a distributed-memory architecture. We only claim that the language supports both architectures equally well.

In summary, Titanium is a language that uses Java as its base, not a strict extension of Java. The Titanium compiler translates Titanium into C, for portability and economy (our prototype is temporarily generating C++). We are not addressing the problem of high performance in the Java Virtual Machine.

In the remainder of the paper we outline the new language features and their relationship to Java, the omissions of Java features, the novel optimizations that are facilitated by the use of Java as a base language, and the driving applications that motivate the initial version of the Titanium language and compiler.

2 Related Work

There are in essence two competing approaches to parallel programming: language and library. We list the most relevant efforts.

Libraries. The relative simplicity and robustness of libraries for parallelism makes them a popular choice for scientific computing. In particular, SPMD programs written in C, C++, or Fortran with MPI for communication and synchronization form the vast majority of large-scale, parallel scientific applications. MPI, however, has a lower raw performance than a global address space. On a Cray T3E, MPI achieves a bandwidth of about 120 MB/sec, while a global address space achieves a bandwidth of about 330 MB/sec [**?**]. Furthermore, with a global address space the compiler can optimize remote accesses with the same techniques used for the local memory hierarchy.

FIDIL. The multidimensional array support in Titanium is strongly influenced by FIDIL *maps* and *domains* [**?**, **?**]. Titanium, however, sacrifices expressiveness for performance. Titanium arrays may only be rectangular, where FIDIL maps have arbitrary shapes. Also, Titanium has two static domain types, general domain and rectangular domain. FIDIL has only a general domain type, thus making it harder to optimize code that uses the more common rectangular kind.

Java-AD. The HPJava project includes Java-AD, an extension of Java for SPMD-style programming [**?**]. The main new feature of Java-AD is multidimensional distributed arrays, similar to HPF arrays. Java-AD also offers an interface to MPI for explicit communication. The current plan is to translate Java-AD into standard Java + MPI calls, This approach prevents optimizations like those we are implementing in Titanium.

Split-C. The parallel execution model and global address space support in Titanium are closely related to Split-C [**?**] and AC [**?**]. Titanium shares a common communication layer with Split-C on distributed memory machines, which we have extended as part of the Titanium project to run on shared memory machines. Split-C differs from Titanium in that the default pointer type is local rather than global; a local pointer default simplifies interfacing to existing sequential code, but a global default makes it easier to port shared memory applications to distributed memory machines. Split-C uses sequential consistency as its default consistency model, but provides explicit operators to allow non-blocking operations to be used. In AC the compiler introduces non-blocking memory operations automatically, using only dependence information, not parallel program analysis. Titanium is closer to AC in this regard.

3 Language Extensions

The main new features of Titanium are immutable classes, explicit support for parallelism, multi-dimensional arrays, and a mechanism for the programmer to control memory management.

```
class Example {
  public static void main (String [])
  {
     // i and n are "replicated variables": they take on
     // the same values in all processes.

     single int n = numberOfIterations();
     for (single int i = 0; i < n; i += 1)
       {
          work1();
          Proc.barrier();
       }
     work2();
     Proc.barrier();
     work3();
  }
}
```

Figure 1: A simple example of synchronization.

3.1 Immutable classes

Java objects are accessed through references. This simplifies the language, but adds a constant overhead (extra level of indirection, object creation and destruction) that reduces the performance of programs that make intensive use of small objects. The typical example is a user-defined complex-number class. To remedy this problem, we introduced *immutable classes*. Immutable classes are not extensions of any existing class (including Object), nor can they be extended. All non-static fields of immutable classes are final.[1] These restrictions allow the compiler to pass such objects by value and to allocate them on the stack or within other objects. In effect, they behave like existing Java primitive types or C *struct*s. Immutable classes are used for some of the special Titanium types such as *Point* (see Section **??**).

3.2 Parallelism

Global synchronization. An important kind of synchronization used in SPMD programs is *global synchronization* in which all processes participate. For instance, a *barrier* causes a process to wait until all other processes reach a barrier. Figure **??** shows a typical SPMD skeleton. A group of processes simultaneously execute the code shown in the figure, synchronizing at the barriers.

The program in Figure **??** is correct as long as all processes have the same value for n: in that

[1]Initialization of such classes relies on the Java 1.1 rule that final fields can be initialized in constructors.

case the barriers ensure that no process executes *work1* while another executes *work2*. If different processes have different values for n, then they execute a different sequence of barrier statements, which is an error. Titanium performs a global-synchronization analysis that ensures, at compile time, that such bugs cannot occur. This global-synchronization analysis is based on recognizing *single-valued* variables: replicated variables (each process owns one instance) that have the same value in all processes. In the example in Figure **??**, i and n must be single-valued. In Titanium, the programmer declares the single-valued variables, and the compiler verifies that the program is *structurally correct*:

> A program is *structurally correct* if all its subexpressions e satisfy the following: Let V be the set of single-valued variables at e. If processes begin execution of e with identical values for each variable in V, and all processes terminate, then all processes execute the same sequence of global synchronization operations and end with identical values for each variable in V.

More details are given by Aiken and Gay [**?**].

Local and global references. The storage associated with a Titanium process is called a *region*. Each object is contained within a single region. Local variables and objects created by **new** are contained in the region of the process that allocates them. References to objects in other regions may be obtained through communication primitives.

By default, all references in Titanium are assumed to be *global*, i.e. they may point to objects in any region. The programmer can declare that a variable v always points to objects of type T in the same region as the current process by declaring that variable with T **local** v. Similarly, the **local** qualifier can be used to declare that an object's field points to an object in the same region. On distributed-memory machines, local pointers are significantly more efficient than global pointers: they take less space, and access to objects is faster. On SMPs global and local pointers are equivalent.

Communication. Processes communicate with each other via reads and writes of object fields, by cloning whole objects, or copying parts of arrays. One-to-all communication is supported by the *broadcast* method, all-to-all communication by the *exchange* method. These two methods also imply a global synchronization of all processes, as all processes must call *broadcast* or *exchange* before the operation can complete. There is also a *barrier* method as defined above. The compiler verifies that uses of these global synchronization operations maintain the structural correctness of the program. Process-to-process synchronization is handled as in Java with *synchronized* methods and the *synchronized* statement.

Consistency model. The programmer may use the synchronization constructs to prevent concurrent reads and writes on shared objects, but there is nothing in the language that rules out race conditions. In some languages that use global address spaces, programmers may use simple shared variables to build synchronization structures such as spinlocks, which appear to the system as shared variables with race conditions. The most intuitive semantics of such shared accesses is *sequential consistency*, which states that memory operations appear to take effect in some total order that is consistent with each processor's program order [?].

On machines with hundreds to thousands of cycles of memory latency for a local or remote memory access, sequential consistency is expensive, and most machine designers have opted for one of the weaker consistency models, such as processor consistency or release consistency. Krishnamurthy and Yelick have shown that languages can provide the stronger model of sequential consistency through static analysis, even when the languages execute on hardware with weaker semantics [?]. We are exploring the use of this analysis, which requires good aliasing and synchronization information, in the context of Titanium, but our current consistency model does not rely on such analysis. Instead, we adopt the Java consistency model, which is weakly consistent at the program level, and roughly says that programmers may see unexpected program behavior unless they protect all conflicting variable accesses with barriers, locks, or other language-specified synchronization primitives.

3.3 Memory management

Titanium incorporates *zone-based* memory management as an extension to Java. Each allocation can be placed in a specific *zone*. Whole zones are freed at once with an explicit *delete-zone* operation. The run-time system maintains reference counts to ensure that zones are not deleted while outstanding references remain. This approach has two advantages: it allows explicit programming for locality, by placing objects that are accessed together in the same zone; and it should perform better than garbage collection on distributed-memory machines. Keeping a reference count on zones rather than individual objects mitigates the problems of cyclical data structures: cyclical data structures can still be reclaimed as long as they are contained in a single zone.

A preliminary study of this style of memory management on sequential C programs found that zones were faster than malloc/free and conservative garbage collection in most cases [?]. This study used a C compiler modified to perform reference counting on all pointers into zones. In a related study, Stoutamire obtained a 13% speedup in an AMR code by using zones to improve data locality [?].

3.4 Arrays, points, and domains

Titanium arrays, which are distinct from Java arrays, are multi-dimensional. They are constructed using *domains*, which specify their index sets, and are indexed by *points*, instead of explicit lists of integers as in Fortran. Points are tuples of integers and domains are sets of points. The following code fragment shows how a multi-dimensional array can be constructed.

```
Point<2> l = [1, 1];
Point<2> u = [10, 20];
RectDomain<2> r = [l : u];
double [2d] A = new double[r];
```

The (two-dimensional) points `l` and `u` are declared and initialized. Then the rectangular domain `r` is initialized to be the set of all points in the rectangle with corners `l` and `u`. Finally the variable `A` is initialized to a two-dimensional array that maps each point of `r` into a **double**.

A multi-dimensional iterator called **foreach** allows one to operate on the elements of `A`. The following **foreach** statement executes its body with `p` bound to successive points of the domain of `A`.

```
foreach (p in A.domain()) {
    A[p] = 42;
}
```

This style of iterator simplifies the removal of array bound checks from the loop body. The iteration order is not specified, which allows the compiler to reorder iterations without the sophisticated and often fragile analysis used by Fortran or C compilers to perform tiling or other optimizations. Titanium's **foreach** is intended to enable these uniprocessor optimizations, not generate parallelism.

Titanium has no array elementwise operators, which would allow writing statements such as `A = B + C`. While more expressive than Titanium's foreach, complex array-level expressions are considerably more difficult to optimize.

The `RectDomain<`N`>` type represents conventional rectangular index ranges in N dimensions. There is also a more general `Domain<`N`>` type, which represents arbitrary index sets. The domain of arrays may only be rectangular, but **foreach** also accepts general domains. This is an important feature for modern partial differential equation solvers, and one that would be hard to implement efficiently with available abstraction mechanisms.

3.5 Other extensions

The preceding extensions were principally motivated by considerations of performance, especially parallel performance. We have placed a lower priority on extensions to enhance expressiveness, but have introduced two. First, Titanium allows programmers to provide additional overloadings of the standard operators and of the array indexing syntax by defining member functions with special names. The motivation here is that we expect such notations to be of interest to our particular audience—scientific programmers. Second, we plan to introduce some form of parameterized types. Here, we are following the on-going debate over the possibility of adding such a facility to Java. If there is a timely resolution, we intend to conform as closely as possible to the selected specification.

4 Incompatibilities

Ideally, Titanium should be (and largely is) a superset of Java for simplicity and for compatibility with existing code. However, there are are two areas of incompatibility: threads and numerics.

- **Threads.** The current version of the Titanium language, and its current implementation, do not support threads. However, we are considering adding threads to the current SPMD model, not for the purpose of parallel execution on multiple processors, but to overlap long-latency operations, such as disk or user I/O. This would also make it possible to use Java modules written with threads (such as the AWT) in Titanium programs. The addition of threads would complicate global synchronization such as barriers, but we believe that the associated analysis can be extended for a limited thread model.

- **Numerics.** Titanium does not adhere to the Java standard for numerics (neither do, we believe, several existing Java implementations). We would like to include support for finer control over IEEE-Floating Point features, such as exceptions and rounding modes, but we have not yet given adequate attention to these issues.

5 Optimizations

Our prototype compiler performs standard analyses and optimizations, such as finding "defs" and "uses" of variables, computing all possible control flow paths, finding and moving loop invariant expressions, finding induction variables, and performing strength reduction of array index expressions. It also omits the construction of the control variable of a `foreach` if, after optimizations, that variable is not necessary. Although we are generating C code and rely on the C compiler to perform certain optimizations, our analysis is done with special knowledge of the Titanium language and libraries. Furthermore, our analyses are able to take advantage of the safe and clean semantics of Java and Titanium. Thus we perform optimizations that we cannot reasonably expect C compilers to perform on the code we produce. Indeed, our experiments have shown that we must perform many additional relatively straightforward optimizations in our compiler if we want them to happen at all.

5.1 Loops

A naive implementation of the loop in Figure **??**a yields poor results. Depending on the static type of `R`, the iteration is over either a `RectDomain` or a `Domain` (whose internal representation is currently a union of `RectDomains`). For simplicity, then, consider iteration over a single `RectDomain`. The address calculation for `A[p]` requires a single pointer increment per iteration of the innermost loop. However, even in this trivial example, most C compilers will not perform strength reduction on the address calculations required by Titanium arrays, which can have arbitrary stride. When we do our own strength reduction we generate the code shown in Figure **??**b. This is an example of what we do to allow Titanium programs to achieve performance competitive with C or Fortran.

6 Applications

Properly evaluating a programming language requires using it to implement representative applications in its intended domain. We are developing the Titanium compiler and run-time system in parallel with non-trivial applications, some ported and some written from scratch. This section describes two of these applications, AMR3D and EM3D.

```
foreach (p in R) {
    A[p] = 42;
}
```

(a)

```
...
for (i_0 = is_0; ; SR_0 += dSR_0) {
  GP_jdouble SR_1 = SR_0;
  for (i_1 = is_1; ; SR_1 += dSR_1) {
    GP_jdouble SR_2 = SR_1;
    for (i_2 = is_2; ; SR_2 += dSR_2) {
      ASSIGN(SR_2, 42.0);
      if ((i_2 += id_2) >= ie_2) break;
    }
  if ((i_1 += id_1) >= ie_1) break;
  }
if ((i_0 += id_0) >= ie_0) break;
}
```

(b)

Figure 2: (a) Titanium code, (b) Excerpt of generated C code for 3D case

6.1 AMR3D

AMR3D is a full 3-dimensional adaptive mesh refinement Poisson solver. The complete program consists of about two thousand lines of Titanium code. Almost half of the code belongs to a routine called the *grid generator*.

AMR is an extension of the multigrid algorithm for linear solvers. Multigrid is a relaxation method that uses grids at different resolutions covering the entire problem domain. AMR allows grids at high levels of resolution to cover only a subset of the problem domain. The area of interest at each level is covered by a set of rectangular *patches*. Similarly to most parallel AMR solvers, we distribute patches across processors, and relax on them in lockstep. If the patches are large enough, communication and synchronization overheads are small.

At various points of the computation, the grid generator recomputes the patch hierarchy based on the need for accuracy. It also load-balances the computation, by assigning a similar amount of work to each processor.

Aside from the grid generator, a large fraction of the code is dedicated to computing boundary values at the interfaces between coarse and fine boundaries. This computation, although not onerous, is complex, and our linguistic support is an effective aid.

AMR3D is the first large Titanium program, and it is interesting to note that the global synchronization analysis (Section **??**) helped uncover a few bugs during its development.

6.2 EM3D

EM3D is the computational kernel from an application that models the propagation of electro-magnetic waves through objects in three dimensions [**?**]. A preprocessing step casts the problem into a simple computation on an irregular bipartite graph containing nodes that represent electric and magnetic field values. The computation consists of a series of "leapfrog" integration steps: on alternate half time steps, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and *vice versa.*

7 Preliminary Results

At this time we can run the programs listed above and several others, but the implementation of our planned set of optimizations is still incomplete.

7.1 Sequential Performance

The first set of benchmarks show the sequential performance of Titanium code. For these experiments, we use a 2D and 3D multigrid Poisson solver, the EM3D computation described above, and the standard DAXPY operation on vectors. The multigrid examples are compared to code written by others in combined C++ and Fortran, while the EM3D and DAXPY examples are compared to C. We were unable to obtain a full 3D AMR code written in another language that performed the same

	C/C++/Fortran	Java Arrays	Titanium Arrays	Overhead
DAXPY	1.4s	6.8s	1.5s	7%
3D multigrid	12s	—	26s	117%
2D multigrid	5.4s	—	6.2s	15%
EM3D	0.7s	1.8s	1.0s	42%

Figure 3: Performance of sequential Titanium compared to other languages.

computation, so this benchmark is not used for sequential comparisons. For the multigrid problems, the 2D grid has 1024×1024 points, and the 3D grid has $64\times64\times64$. The EM3D graph is synthetic with 500 nodes, fixed degree 20, and random connectivity. The DAXPY is a 100K element vector. All of these numbers were taken on a 166Mhz Ultrasparc processor.

We present the performance of two versions of the DAXPY and EM3D problems, one that uses standard Java arrays and another using Titanium arrays with bounds checking turned off. (The multigrid solvers make extensive use of our domain operators, and we have not written them in pure Java.) The Java array version can be compiled with a standard Java compiler. We do not have access to a native code Java compiler for this machine, and interpreted Java byte codes are roughly one order of magnitude slower than Titanium. The last column in the table below shows the percent increase in running time of the Titanium array version relative to the C or C++/Fortran version.

7.2 Parallel Performance

Titanium runs on top of a standard Posix thread library on SMPs and on Active Message layer [**?**] on distributed memory multiprocessors and networks of workstations. Figure **??** shows the speedup of EM3D and AMR3D on an 8-way SUN Enterprise SMP for a fixed problem size. The performance shows that the overhead for our parallel runtime library are minimal. AMR3D is still under development, and we are running it on a shallow grid hierarchy (2 levels). Its speedup is limited primarily by the serial multigrid computation on the coarse grid. EM3D attains almost linear speedups as the runtime overheads are offset by improved cache behavior on smaller data sets.

8 Conclusions

Our experience thus far is that Java is a good choice as a base language: it is easy to extend, and its safety features greatly simplify the compiler writer's task. We also believe that extending Java is easier than obtaining high performance within Java's strict language specs (assuming that the latter is at all feasible). Many of the features of Titanium would be hard or impossible to achieve as Java libraries, and the compiler would not be able to perform static analysis and optimizations on them.

We have several goals for the project. We wish to make the system robust and available for use in the scientific computing community. We also wish to use it as a basis for research on optimization of explicitly parallel programs, optimizations for memory hierarchy, and domain-specific language extensions.

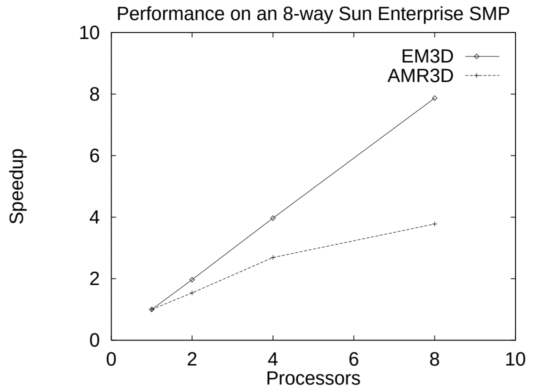

Figure 4: Speedups on an Sun Ultrasparc SMP

Acknowledgements. We thank Ben Greenwald and Joe Darcy for their earlier contributions to Titanium. We also thank Intel for various hardware donations, including the Millennium grant, and Sun Microsystems for donating the NOW and Clumps multiprocessor hardware.