Titanium: A High Performance Java Dialect

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

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

Extended Abstract

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.

Titanium is a parallel SPMD (for Single Program, Multiple Data) dialtect of Java. 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.

The starting design point for Titanium is Java. We have chosen Java for several reasons. First, Java is a relatively small and clean object-oriented language (compared, for instance, to C++) and therefore is easy to extend. Second, Java is popular and is itself based on popular languages (C/C++). Learning Titanium requires little effort for those acquainted with Java-like languages. Also, 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. Our modifications to Java have been influenced by our primary goals of performance, safety, and expressiveness. In this presentation, we focus primarily on Titanium's support for enhancing performance.

Performance is a fundamental requirement for computationally demanding scientific applications. Many design choices reflect this goal. For instance, the execution model is explicitly parallel SPMD, thus eliminating the need for a parallelizing compiler as well as avoiding the overhead of mapping a large number of threads to a smaller fixed number of processors. 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 (similar to the annotations in Split-C) that the compiler may find hard to infer reliably from static analysis. Also, the Titanium compiler translates Titanium into C to avoid incurring the costs of interpreted execution as well as enjoy the benefits of some of the traditional sequential optimizations provided by C compilers. Finally, Titanium has built-in features such as true multi-dimensional arrays and iterators, points and index sets (including irregular ones) as first-class values. We discuss briefly the language support for multi-dimensional arrays.

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];
```

	SPARC					Pentium			
	Titanium	JIT	gcc	cc	f77	Titanium	gcc	cc	f77
DAXPY	1.6	3.3	1.4	1.9	1.6	2.3	1.8	1.8	1.9
EM3D	1.7	3.5	0.89	1.0		1.6	0.95	0.86	
2D Multigrid	5.0				5.3	5.5			7.3
3D Multigrid	22				12	20			23

Figure 0.1: Performance of sequential Titanium compared to versions in other languages. All numbers are in seconds. All compilers were given the -O4 flag. Output of the Titanium compiler was compiled with gcc.

```
RectDomain<2> r = [l : u];
double [2d] A = new double[r];
foreach (p in A.domain())
    A[p] = 42;
```

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`.

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.

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, which is an important feature for modern partial differential equation solvers.

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.

Preliminary Results: Our benchmarks show that the sequential performance of Titanium code is comparable to performance obtained by C and Fortran versions. For these experiments, we use a 2D and 3D multigrid Poisson solver, an application kernel that models electromagnetics waves (EM3D), 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 a Java version executed using Sun's Just-In-Time compiler (JIT) as well as versions written in C and/or Fortran. 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 used a 100K element vector. Data were taken on a 166Mhz UltraSPARC processor and a 200Mhz Pentium Pro running Solaris. The cc and f77 compilers are from Sun. The output of the Titanium compiler is fed to the GNU C compiler.

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 programmer's task. We also believe that extending Java is easier than obtaining high performance within Java's strict language specifications (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.