UC Berkeley

UC Berkeley Previously Published Works

Title

Titanium

Permalink

https://escholarship.org/uc/item/60v4m2ph

ISBN

9780387097657

Authors

Yelick, K
Graham, SL
HILFINGER, P
et al.

Publication Date

2011

DOI

10.1007/978-0-387-09766-4_516

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Titanium

Katherine Yelick[1], Susan L. Graham[2], Paul Hilfinger[2], Dan Bonachea[3], Jimmy Su[2], Amir Kamil[2], Kaushik Datta[2], Phillip Colella[2], Tong Wen[2]
[1]University of California at Berkeley and Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]University of California, Berkeley, CA, USA
[3]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Definition

Titanium is a parallel programming language designed for high-performance scientific computing. It is based on Java™ and uses a Single Program Multiple Data (SPMD) parallelism model with a Partitioned Global Address Space (PGAS).

Discussion

Introduction

Titanium is an explicitly parallel dialect of Java™ designed for high-performance scientific programming [14, 15]. 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 that would enable portable programming for a wide range of parallel platforms by striking an appropriate balance between expressiveness, user-provided information about concurrency and memory locality, and compiler and run-time support for parallelism. The 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 was to build on the experience of several Partitioned Global Address Space (PGAS) languages, 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. Titanium uses Java as the underlying base language, but is neither a strict superset nor subset of that language. Titanium adds general multidimensional arrays, support for extending the value types in the language, and an unordered loop construct. In place of Java threads, which are used for both program structuring and concurrency, Titanium uses a static thread model with a partitioned address space to allow for locality optimizations.

Titanium's Parallelism Model

Titanium uses a Single Program Multiple Data (SPMD) parallelism model, which is familiar to users of message-passing models. The following simple Titanium program illustrates the use of built-in methods **Ti.numProcs()** and **Ti.thisProc()**, which query the environment for the number of threads (or processes) and the index within that set of the executing thread. The example prints these indices in arbitrary order. The number of Titanium threads need not be equal to the number of physical processors, a feature that is often useful when debugging parallel code on single-processor machines. However, high-performance runs typically use a one-to-one mapping between Titanium threads and physical processors.

```
class HelloWorld {
  public static void main (String [] argv) {
```

```
    System.out.println("Hello from proc " +
      Ti.thisProc() + " out of " + Ti.numProcs());
  }
}
```

Titanium supports Java's synchronized blocks, which are useful for protecting asynchronous accesses to shared objects. Because many scientific applications use a bulk-synchronous style, Titanium also has 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 Ti- tanium's parallel execution model is that barriers must be textually aligned in the program – not only must all threads reach a barrier before any one of them may proceed, but they must all reach the same textual barrier. For example, the following program is not legal in Titanium:

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

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

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

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

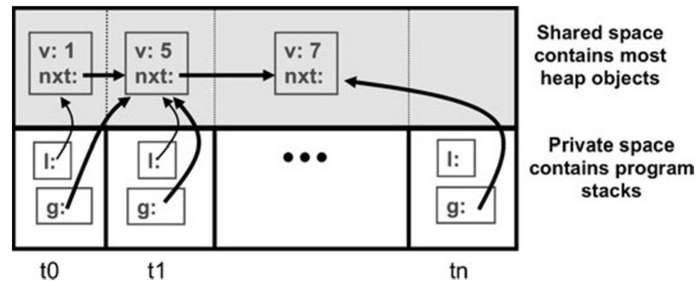
```
int single allTimestep = 0;
int single allEndTime = broadcast
   inputTimeSteps from 0;
for (; allTimestep < allEndTime;
   allTimestep)++{
  < read values belonging to other threads >
  Ti.barrier();
  < compute new local values >
  Ti.barrier();
}
```

Barrier analysis is entirely static and provides compile-time prevention of barrier-based deadlocks. It can also be used to improve the quality of concurrency analysis used in optimizations. 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.

Titanium's Memory Model

The two basic mechanisms for communicating between threads are accessing shared variables and sending messages. Shared memory is generally considered easier to program, because communication is one-sided: Threads can access shared data at any time without interrupting other threads, and shared data structures can be directly represented in memory. Titanium is based on a Partitioned Global Address Space (PGAS) model, which is similar to shared memory but with an explicit recognition that access time is not uniform. As shown in Fig. 1, memory is partitioned such that each partition has affinity to one thread. Memory is also partitioned orthogonally into private and shared memory, with stack variables living in private memory, and heap objects, by default, living in the shared space. A thread may access any variable that resides in shared space, but has fast access to variables in its own partition. Objects created by a given thread will reside in its own part of the memory space.

Titanium statically makes an explicit distinction between local and global references: A local reference must refer to an object within the same thread partition, while a global reference may refer to either a remote or



Titanium. Fig. 1 Titanium's partitioned global address space memory model

local partition. In Fig. 1, instances of **l** are local references, whereas **g** and **nxt** are global references and can therefore cross partition boundaries. The motivation for this distinction is performance. Global references are more general than local ones, but they often incur a space penalty to store affinity information and a time penalty upon dereference to check whether communication is required. References in Titanium are global by default, but may be designated local using the local type qualifier. The compiler performs type inference to automatically label variables as local [10].

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

Titanium Arrays

Java arrays do not support sub-array objects that are shared with larger arrays, nonzero base indices, or true multidimensional arrays. Titanium retains Java arrays for compatibility, but adds its own multidimensional array support, 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 a language for Finite Different Calculations, FIDIL, designed by Colella and Hilfinger [7]. 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, NAS multigrid (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 $-1 \leqslant i, j, k \leqslant 256$. 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**. Titanium arrays may start at an arbitrary base point, as the example with a $[-1,\ -1,\ -1]$ base shows. In this example, the grid was designed to have space for ghost regions, which are

all the points that have either −1 or 256 as a coordinate. 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), though they are accessible from any process in the parallel program.

The power of Titanium arrays stems from array operators that can be used to create alternative views of an array's data, 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.

Domain Calculus

Titanium's domain calculus operators support subarrays both syntactically and from a performance standpoint. The tedious business of index calculations and array offsets has been migrated from the application code to the compiler and runtime system. For example, 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 without creating a copy
double [3d] gridAInterior = gridA.shrink(1);
//update overlapping ghost cells
   from neighboring block
//by copying values from gridA to gridB
gridB.copy(gridAInterior);
```

The above example appears in a NAS MG implementation in Titanium [4], except that **gridA** and **gridB** are themselves elements of a higher-level array structure. The copy operation as it appears here performs contiguous or noncontiguous memory copies, and may perform interprocessor communication when the two grids reside in different processor memory spaces. 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.

Unordered Loops, Value Types, and Overloading

The *foreach* construct provides an unordered looping construct designed for iterating through a multidimensional space. In the foreach loop below, the point **p** plays the role of a loop index variable.

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

The **applyStencil** method may safely refer to elements that are one point away from **p**, since the loop is over the interior of a larger array.

This one loop concisely expresses an iteration over a multidimensional domain that would correspond to a multi-level loop nest in other languages. A common class of loop bounds and indexing errors is avoided by having the compiler and runtime system automatically manage the iteration boundaries for the multidimensional traversal. 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 domain explicitly allows the compiler to reorder loop iterations to maximize performance – for instance, by performing automatic cache blocking and tiling optimizations [12]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

The Titanium immutable class feature provides language support for defining application-specific primitive types (often called "lightweight" or "value" classes), allowing the creation of user-defined unboxed objects, analogous to C structs. Immutables provide efficient support for extending the language with new types which are manipulated and passed by value, avoiding pointer-chasing overheads which would otherwise be associated with the use of tiny objects in Java.

One compelling example of the use of immutables is for defining a Complex number class, which was used in a Titanium implementation of the NAS FT benchmark.

Titanium also allows for operator overloading, a feature that was strongly desired by application developers on the team, and was used in the FT example to simplify the expressions on complex values.

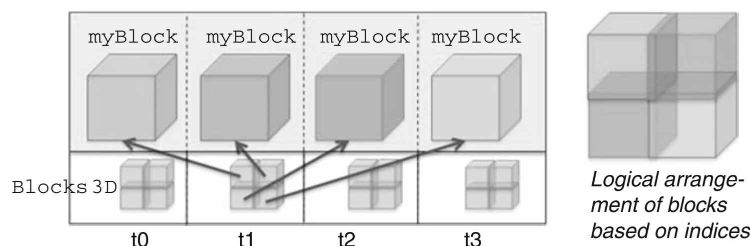
Distributed Arrays

Titanium also supports the construction of distributed array data structures, which are built from local pieces rather than declared as distributed types. This reflects the design emphasis on adaptive and sparse data structures in Titanium, rather than the simpler "regular array" computations that could be supported with simpler flat arrays. The general pointer-based distribution mechanism combined with the use of arbitrary base indices for arrays provides an elegant and powerful mechanism for shared data.

The following code is a portion of the parallel Titanium code for a multigrid computation. It is run on every processor and creates the **blocks3D distributed** array, which can access any processor's portion of the grid.

```
Point< 3 > startCell =
myBlockPos * numCellsPerBlockSide;
Point< 3 > endCell = startCell + (numCellsPerBlock
  Side - [1,1,1]);
double [3d] myBlock =
new double[startCell:endCell];
//"blocks" is used to create "blocks3D" array
double [1d] single [3d] blocks =
  new double [0:(Ti.numProcs()-1)] single [3d];
blocks.exchange(myBlock);
//create local "blocks3D" array
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)];
```

Each processor computes its start and end indices by performing arithmetic operations on **Points**. These indices are used to create a local **myBlock** array. Every processor also allocates its own 1D array **blocks**. Then, by combining the **myBlock** arrays using the exchange operation, **blocks** becomes a distributed data structure. As shown in Fig. 2, the exchange operation performs an all-to-all broadcast and stores each processor's contribution in the corresponding element of its local blocks array. To create a more natural mapping, a 3D processor array is used, with each element containing a reference to a particular local block. By using global indices in the local block – meaning that each block has a different set of indices that overlap only in the area of ghost regions – the copy operations described above can be used to update the ghost cells. The generality of Titanium's distributed data structures is not fully utilized in the example of a uniform mesh, but in an adaptive block structured mesh, a union of rectangles can be used to



Titanium. Fig. 2 Distributed 3D array in titanium's PGAS address space. The pointers in the blocks3D array are shown only for thread t1 for simplicity

fill a spatial area, and the global indexing and global address space used to simplify much more complicated ghost region updates.

Implementation Techniques and Research

The Titanium compiler translates Titanium code into C code, and then hands that code off to a C compiler to be compiled and linked with the Titanium runtime system and, in the case of distributed memory back ends, with the GASNet communication system [5]. The choice of C as a target was made to achieve portability, and produces reasonable performance without the overhead of a virtual machine. GASNet is a one-sided communication library that is used within a number of other PGAS language implementations, including Co-Array Fortran, Chapel, and multiple UPC implementations. GASNet is itself designed for portability, and it runs on top of Ethernet (UDP) and MPI, but there are optimized implementations for most of the high-speed networks that are used in clusters and supercomputers designs. Titanium can also run on shared memory systems using a runtime layer based on POSIX Threads, and on combinations of shared and distributed memory by combining this with GASNet. Titanium, like Java, is designed for memory safety, and the Titanium runtime system includes the Boehm-Weiser garbage collector for shared memory code. To handle distributed memory environments, the runtime system tracks references that leak to remote nodes, but also adds a scalable region-based memory management concept to the language along with compiler analysis [5].

Aggressive program analysis is crucial for effective optimization of parallel code. In addition to serial loop optimizations [12] and some standard optimizations to reduce the size and complexity of generate C code, the compiler performs a number of novel analyses on parallelism constructs. For example, information about what sections of code may operate concurrently is useful for many optimizations and program analyses. In combination with alias analysis, it allows the detection of potentially erroneous race conditions, the removal of unnecessary synchronization operations, and the ability to provide stronger memory consistency guarantees. Titanium's textually aligned barriers divide the code into independent phases, which can be exploited to improve the quality of concurrency analysis. The single-valued expressions are also used to improve concurrency analysis on branches. These two features 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.

Alias analysis identifies pointer variables that may, must, or cannot reference the same object. The Titanium compiler uses alias analysis to enable other analyses (such as locality and sharing analysis), and 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. The Titanium compiler's alias analysis is a Java derivative of Andersen's points-to analysis with extensions to handle multiple threads. The modified analysis is only a constant factor slower than the sequential analysis, and its running time is independent of the number of runtime threads.

Application Experience

A number of benchmarks and larger applications have been written in Titanium, starting with some of the NAS Benchmarks [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 [2]. 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. The most ambitious efforts have been applications frameworks for Adaptive Mesh Refinement (AMR) algorithms and Immersed Boundary Method simulations [6] by Tong Wen and Ed Givelberg, respectively. 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 by Luigi Semenzato, AMR gas dynamics [11] by Peter McCorquodale and Immersed Boundaries for simulation of the heart by Armando Solar-Lezama and cochlea by Ed Givelberg, with various optimization and analysis efforts by Sabrina Merchant, Jimmy Su, and Amir Kamil.

The performance results show good scalability on the applications problems on up to hundreds of separate distributed memory nodes, and performance that is in some cases comparable to applications written in C++ or FORTRAN with message passing. The compiler is a research prototype and does not have all of the static and dynamic optimizations one would expect from a commercial compiler, but even serial running-time comparisons show competitive performance. No formal productivity studies involving humans have been done, but a variety of case studies have shown that the global address space combined with a powerful multi-dimensional array abstraction and the data abstraction support derived from Java leads to code that is elegant and concise.

Related Entries

►Coarray Fortran
►PGAS (Partitioned Global Address Space) Languages
►UPC

Bibliography

1. Aiken A, Gay D (1998) Barrier inference. In: Principles of programming languages, San Diego, CA
2. Balls GT, Colella P (2002) A finite difference domain decomposition method using local corrections for the solution of Poisson's equation. J Comput Phys 180(1):25–53
3. Bonachea D (2002) GASNet specification. Technical report CSD-02-1207, University of California, Berkeley
4. Datta K, Bonachea D, Yelick K (2005) Titanium performance and potential: an NPB experimental study. In: 18th international workshop on languages and compilers for parallel computing (LCPC). Hawthorne, NY, October 2005
5. Gay D, Aiken A (2001) Language support for regions. In: SIGPLAN conference on programming language design and implementation. Washington, DC, pp 70–80
6. Givelberg E, Yelick K Distributed immersed boundary simulation in titanium. http://titanium.cs.berkeley.edu, 2003
7. Hilfinger PN, Colella P (1989) FIDIL: a language for scientific processing. In: Grossman R (ed) Symbolic computation: applications to scientific computing. SIAM, Philadelphia, pp 97–138
8. Kamil A, Yelick K (2007) Hierarchical pointer analysis for distributed programs. Static Analysis Symposium (SAS), Kongens Lyngby, Denmark, August 22–24, 2007
9. Kamil A, Yelick K (2010) Enforcing textual alignment of collectives using dynamic checks. In: 22nd international workshop on languages and compilers for parallel computing (LCPC), October 2009. Also appears in Lecture notes in computer science, vol 5898. Springer, Berlin, pp 368–382. DOI: 10.1007/978-3-642-13374-9
10. Liblit B, Aiken A (2000) Type systems for distributed data structures. In: The 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), Boston, January 2000
11. McCorquodale P, Colella P (1999) Implementation of a multilevel algorithm for gas dynamics in a high-performance Java dialect. In: International parallel computational fluid dynamics conference (CFD'99)
12. Pike G, Semenzato L, Colella P, Hilfinger PN (1999) Parallel 3D adaptive mesh refinement in Titanium. In: 9th SIAM conference on parallel processing for scientific computing, San Antonio, TX, March 1999
13. Su J, Yelick K (2005) Automatic support for irregular computations in a high-level language. In: 19th International Parallel and Distributed Processing Symposium (IPDPS)
14. Yelick K, Hilfinger P, Graham S, Bonachea D, Su J, Kamil A, Datta K, Colella P, Wen T (2007) Parallel languages and compilers: perspective from the titanium experience. Int J High Perform Comput App 21:266–290
15. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998) Titanium: a high-performance Java dialect. Concur: Pract Exp 10:825–836

Web Documentation Bibliography

GASNet Home Page. http://gasnet.cs.berkeley.edu/
Titanium Project Home Page at http://titanium.cs.berkeley.edu.