Titanium: A High-Productivity, High-Performance Language

Amir Kamil and Kathy Yelick
Lawrence Berkeley National Laboratory and UC Berkeley

Joint work with

Motivation: Target Problems

• Many modeling problems in astrophysics, biology, material science, and other areas require
  • Enormous range of spatial and temporal scales

• To solve interesting problems, one needs:
  • Complex data structures
  • Adaptive methods
  • Large scale parallel machines

• Titanium is designed for
  • Structured grids
  • Locally-structured grids (AMR)

• Source: J. Bell, LBNL
Titanium Background

- Based on Java
  - Classes, automatic memory management, etc.
  - Compiled to C and then machine code, no JVM
- Same parallelism model as UPC and CAF
  - SPMD parallelism
  - Dynamic Java threads are not supported
- Optimizing compiler
  - Analyzes global synchronization
  - Optimizes pointers, communication, memory
Summary of Features Added to Java

- Multidimensional arrays: iterators, subarrays, copying
- Immutable ("value") classes
- Templates
- Operator overloading
- Scalable SPMD parallelism replaces threads
- Global address space with local/global reference distinction
- Checked global synchronization
- Zone-based memory management (regions)
- Libraries for collective communication, distributed arrays, bulk I/O, performance profiling
Outline

• Titanium Execution Model
  • SPMD
  • Global synchronization
  • Single

• Titanium Memory Model
• Support for Serial Programming
• Performance and Applications
• Hierarchical Execution Model
• Hierarchical Memory Model
SPMD Execution Model

- Titanium has the same execution model as UPC and CAF
- Basic Java programs may be run as Titanium programs, but all processors do all the work.
- E.g., parallel hello world
  ```java
class HelloWorld {
    public static void main (String [] argv) {
      System.out.println("Hello from proc "+ Ti.thisProc() + " out of "+ Ti.numProcs());
    }
  }
```
- Global synchronization done using `Ti.barrier()`
Barriers and Single

- Common source of bugs is barriers or other collective operations inside branches or loops
  - barrier, broadcast, reduction, exchange
- A “single” method is one called by all procs
  - public single static void allStep(...) 
- A “single” variable has same value on all procs
  - int single timestep = 0;
- Single annotation on methods is optional, but useful in understanding compiler messages
- Compiler proves that all processors call barriers together
Explicit Communication: Broadcast

- Broadcast is a one-to-all communication
  
  `broadcast <value> from <processor>`

- For example:
  ```
  int count = 0;
  int allCount = 0;
  if (Ti.thisProc() == 0) count = computeCount();
  allCount = broadcast count from 0;
  ```

- The processor number in the broadcast must be single;
  all constants are single.
  - All processors must agree on the broadcast source.
  - The `allCount` variable could be declared single.
    - All will have the same value after the broadcast.
More on Single

• Global synchronization needs to be controlled
  if (this processor owns some data) {
    compute on it
    barrier
  }

• Hence the use of “single” variables in Titanium

• If a conditional or loop block contains a barrier, all processors must execute it
  • conditions must contain only single variables

• Compiler analysis statically enforces freedom from deadlocks due to barrier and other collectives being called non-collectively "Barrier Inference" [Gay & Aiken]
Single Variable Example

- Barriers and single in N-body Simulation

```java
class ParticleSim {
    public static void main (String [] argv) {
        int single allTimestep = 0;
        int single allEndTime = 100;
        for (; allTimestep < allEndTime; allTimestep++){
            read remote particles, compute forces on mine
            Ti.barrier();
            write to my particles using new forces
            Ti.barrier();
        }
    }
}
```

- Single methods inferred by the compiler
Outline

• Titanium Execution Model
• **Titanium Memory Model**
  • Global and local references
  • Exchange: building distributed data structures
  • Region-based memory management
• Support for Serial Programming
• Performance and Applications
• Hierarchical Execution Model
• Hierarchical Memory Model
Global Address Space

- Globally shared address space is partitioned
- References (pointers) are either local or global (meaning possibly remote)
Use of Global / Local

• Global references (pointers) may point to remote locations
  • Reference are global by default
  • Easy to port shared-memory programs

• **Global pointers are more expensive than local**
  • True even when data is on the same processor
  • Costs of global:
    • space (processor number + memory address)
    • dereference time (check to see if local)

• **May declare references as local**
  • Compiler will automatically infer local when possible
  • This is an important performance-tuning mechanism
Global Address Space

- Processes allocate locally
- References can be passed to other processes

class C { public int val;... }
C gv;        // global pointer
C local lv;  // local pointer
if (Ti.thisProc() == 0) {
    lv = new C();
}
gv = broadcast lv from 0;  // data race
gv.val = Ti.thisProc()+1;
int winner = gv.val
Aside on Titanium Arrays

- Titanium adds its own multidimensional array class for performance
- Distributed data structures are built using a 1D Titanium array
- Slightly different syntax, since Java arrays still exist in Titanium, e.g.:
  ```java
  int [1d] a;
  a = new int [1:100];
  ```
- Will discuss these more later...
Explicit Communication: Exchange

- To create shared data structures
  - each processor builds its own piece
  - pieces are exchanged (for objects, just exchange pointers)

- Exchange primitive in Titanium

```java
int [1d] single allData;
allData = new int [0:Ti.numProcs()-1];
allData.exchange(Ti.thisProc()*2);
```

- E.g., on 4 procs, each will have copy of allData:
  ![Diagram of allData distribution across processors]
**Distributed Data Structures**

- Building distributed arrays:

  ```
  Particle [1d] single [1d] allParticle =
  new Particle [0:Ti.numProcs-1][1d];
  Particle [1d] myParticle =
  new Particle [0:myParticleCount-1];
  allParticle.exchange(myParticle);
  ```

- Now each processor has array of pointers, one to each processor’s chunk of particles

  ![Diagram showing all-to-all broadcast between processors P0, P1, and P2](image)
Region-Based Memory Management

• An advantage of Java over C/C++ is:
  • Automatic memory management

• But garbage collection:
  • Has a reputation of slowing serial code
  • Does not scale well in a parallel environment

• Titanium approach “Regions” [Gay & Aiken]:
  • Preserves safety – cannot deallocate live data
  • Garbage collection is the default (on most platforms)
  • Higher performance is possible using region-based explicit memory management
  • Takes advantage of memory management phases
Region-Based Memory Management

- Need to organize data structures
- Allocate set of objects (safely)
- Delete them with a single explicit call (fast)

```java
PrivateRegion r = new PrivateRegion();
for (int j = 0; j < 10; j++) {
    int[] x = new (r) int[j + 1];
    work(j, x);
}
try { r.delete(); }
catch (RegionInUse oops) {
    System.out.println(“failed to delete”);
}
```
Outline

• Titanium Execution Model
• Titanium Memory Model
• **Support for Serial Programming**
  • Immutables
  • Operator overloading
  • Multidimensional arrays
  • Templates
• Performance and Applications
• Hierarchical Execution Model
• Hierarchical Memory Model
Java Objects

- Primitive scalar types: boolean, double, int, etc.
  - implementations store these on the program stack
  - access is fast -- comparable to other languages
- Objects: user-defined and standard library
  - always allocated dynamically in the heap
  - passed by pointer value (object sharing)
  - has implicit level of indirection
  - simple model, but inefficient for small objects

2.6
3
true

real: 7.1
imag: 4.3
Java Object Example

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

Complex c = new Complex(7.1, 4.3);
c = c.add(c);
class VisComplex extends Complex { ... }
Immutable Classes in Titanium

• For small objects, would sometimes prefer
  • to avoid level of indirection and allocation overhead
  • pass by value (copying of entire object)
  • especially when immutable -- fields never modified
    • extends the idea of primitive values to user-defined types

• Titanium introduces immutable classes
  • all fields are implicitly final (constant)
  • cannot inherit from or be inherited by other classes
  • needs to have 0-argument constructor

• Examples: Complex, xyz components of a force
Example of Immutable Classes

• The immutable complex class nearly the same
  immutable class Complex {
    Complex () {real=0; imag=0;}
    ...
  }
  Zero-argument constructor required
  Rest unchanged. No assignment to fields outside of constructors.

• Use of immutable complex values
  Complex c1 = new Complex(7.1, 4.3);
  Complex c2 = new Complex(2.5, 9.0);
  c1 = c1.add(c2);

• Addresses performance and programmability
  • Similar to C structs in terms of performance
  • Support for Complex with a general mechanism
Operator Overloading

• Titanium provides operator overloading
  • Convenient in scientific code
  • Feature is similar to that in C++

```java
class Complex {
    ...
    public Complex op+(Complex c) {
        return new Complex(c.real + real, c.imag + imag);
    }
}
```

Complex c1 = new Complex(7.1, 4.3);
Complex c2 = new Complex(5.4, 3.9);
Complex c3 = c1 + c2;
```
Arrays in Java

- Arrays in Java are objects
- Only 1D arrays are directly supported
- Multidimensional arrays are arrays of arrays
- General, but slow

- Subarrays are important in AMR (e.g., interior of a grid)
  - Even C and C++ don’t support these well
  - Hand-coding (array libraries) can confuse optimizer

- Can build multidimensional arrays, but we want
  - Compiler optimizations and nice syntax
Multidimensional Arrays in Titanium

• New multidimensional array added
  • Supports subarrays without copies
    • can refer to rows, columns, slabs, interior, boundary, even elements…
  • Indexed by Points (tuples of ints)
  • Built on a rectangular set of Points, RectDomain
  • Points, Domains and RectDomains are built-in immutable classes, with useful literal syntax

• Support for AMR and other grid computations
  • domain operations: intersection, shrink, border
  • bounds-checking can be disabled after debugging
Unordered Iteration

- **Motivation:**
  - Memory hierarchy optimizations are essential
  - Compilers sometimes do these, but hard in general

- **Titanium has explicitly unordered iteration**
  - Helps the compiler with analysis
  - Helps programmer avoid indexing details

```
foreach (p in r) { ... A[p] ... }
```

- \( p \) is a Point (tuple of ints), can be used as array index
- \( r \) is a RectDomain or Domain

- **Additional operations on domains to transform**
- **Note:** foreach is not a parallelism construct
Point, RectDomain, Arrays in General

- Points specified by a tuple of ints
  
  ```java
  Point<2> lb = [1, 1];
  Point<2> ub = [10, 20];
  ```

- RectDomains given by 3 points:
  - lower bound, upper bound (and optional stride)
    ```java
    RectDomain<2> r = [lb : ub];
    ```

- Array declared by num dimensions and type
  ```java
  double [2d] a;
  ```

- Array created by passing RectDomain
  ```java
  a = new double [r];
  ```
Simple Array Example

• Matrix sum in Titanium

Point<2> lb = [1,1];
Point<2> ub = [10,20];
RectDomain<2> r = [lb:ub];

double [2d] a = new double [r];
double [2d] b = new double [1:10,1:20];
double [2d] c = new double [lb:ub:[1,1]];

for (int i = 1; i <= 10; i++)
    for (int j = 1; j <= 20; j++)
        c[i,j] = a[i,j] + b[i,j];

foreach(p in c.domain()) { c[p] = a[p] + b[p]; }
More Array Operations

• Titanium arrays have a rich set of operations

  translate restrict slice (n dim to n-1)

• None of these modify the original array, they just create another view of the data in that array

• You create arrays with a RectDomain and get it back later using A.domain() for array A
  • A Domain is a set of points in space
  • A RectDomain is a rectangular one

• Operations on Domains include +, -, * (union, difference, intersection)
public static void matMul(double [2d] a,
        double [2d] b,
        double [2d] c) {
    foreach (ij in c.domain()) {
        double [1d] aRowi = a.slice(1, ij[1]);
        double [1d] bColj = b.slice(2, ij[2]);
        foreach (k in aRowi.domain()) {
            c[ij] += aRowi[k] * bColj[k];
        }
    }
}

Performance: comparable to 3 nested loops in C
Example: Setting Boundary Conditions

`foreach (l in local_grids.domain()) {
    foreach (a in all_grids.domain()) {
        local_grids[l].copy(all_grids[a]);
    }
}

- Can allocate arrays in a global index space
- Let compiler compute intersections
Templates

• Many applications use containers:
  • Parameterized by dimensions, element types,…
  • Java supports parameterization through inheritance
    • Can only put Object types into containers
    • Inefficient when used extensively

• Titanium provides a template mechanism closer to C++
  • Can be instantiated with non-object types (double, Complex) as well as objects

• Example: Used to build a distributed array package
  • Hides the details of exchange, indirection within the data structure, etc.
Example of Templates

template <class Element> class Stack {
    . . .
    public Element pop() {...}
    public void push( Element arrival ) {...}
}

Stack<int> list = new Stack<int>();
list.push( 1 );  // Not an object
int x = list.pop();  // Strongly typed, No dynamic cast

• Addresses programmability and performance
Using Templates: Distributed Arrays

template <class T, int single arity>
public class DistArray {
    RectDomain <arity> single rd;
    T [arity d][arity d] subMatrices;
    RectDomain <arity> [arity d] single subDomains;
    ...
    /* Sets the element at p to value */
    public void set (Point <arity> p, T value) {
        getHomingSubMatrix (p) [p] = value;
    }
}

DistArray <double, 2> single A =
    new DistArray<double, 2> ( [[0,0]:[aHeight, aWidth]] );
Outline

• Titanium Execution Model
• Titanium Memory Model
• Support for Serial Programming
• Performance and Applications
  • Serial performance on pure Java (SciMark)
  • Parallel applications
  • Compiler status & usability results
• Hierarchical Execution Model
• Hierarchical Memory Model
Java Compiled by Titanium Compiler

- Sun JDK 1.4.1_01 (HotSpot(TM) Client VM) for Linux
- IBM J2SE 1.4.0 (Classic VM cxia32140-20020917a, jtc JIT) for 32-bit Linux
- Titaniumc v2.87 for Linux, gcc 3.2 as backend compiler -O3. no bounds check
- gcc 3.2, -O3 (ANSI-C version of the SciMark2 benchmark)
- Same as previous slide, but using a larger data set
  - More cache misses, etc.
- Performance of IBM/Java and Titanium are closer to, sometimes faster than C.
Applications in Titanium

• Benchmarks and Kernels
  • Scalable Poisson solver [Balls & Colella]
  • NAS PB: MG, FT, IS, CG [Datta & Yelick]
  • Unstructured mesh kernel: EM3D
  • Dense linear algebra: LU, MatMul [Yau & Yelick]
  • Tree-structured n-body code
  • Finite element benchmark

• Larger applications
  • Gas Dynamics with AMR [McQuorquodale & Colella]
  • Heart & Cochlea simulation [Givelberg, Solar, Yelick]
  • Genetics: micro-array selection [Bonachea]
  • Ocean modeling with AMR [Wen & Colella]
Case Study: Block-Structured AMR

- Adaptive Mesh Refinement (AMR) is challenging
  - Irregular data accesses and control from boundaries
  - Mixed global/local view is useful

Titanium AMR benchmarks available

AMR Titanium work by Tong Wen and Philip Colella
AMR in Titanium

C++/Fortran/MPI AMR
- Chombo package from LBNL
- Bulk-synchronous comm:
  - Pack boundary data between procs

Titanium AMR
- Entirely in Titanium
- Finer-grained communication
  - No explicit pack/unpack code
  - Automated in runtime system

<table>
<thead>
<tr>
<th>Code Size in Lines</th>
<th>C++/Fortran/MPI</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR data Structures</td>
<td>35000</td>
<td>2000</td>
</tr>
<tr>
<td>AMR operations</td>
<td>6500</td>
<td>1200</td>
</tr>
<tr>
<td>Elliptic PDE solver</td>
<td>4200*</td>
<td>1500</td>
</tr>
</tbody>
</table>

10X reduction in lines of code!

* Somewhat more functionality in PDE part of Chombo code

Work by Tong Wen and Philip Colella; Communication optimizations by Jimmy Su and Kathy Yelick
**Performance of Titanium AMR**

- **Serial:** Titanium is within a few % of C++/F; sometimes faster!
- **Parallel:** Titanium scaling is comparable with generic optimizations
  - additional optimizations (namely overlap) not yet implemented
Immersed Boundary Simulation in Titanium

- Modeling elastic structures in an incompressible fluid.
  - Blood flow in the heart, blood clotting, inner ear, embryo growth, and many more
- Complicated parallelization
  - Particle/Mesh method
  - “Particles” connected into materials

<table>
<thead>
<tr>
<th>Code Size in Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
</tr>
<tr>
<td>8000</td>
</tr>
</tbody>
</table>

Work by Ed Givelberg, Armando Solar-Lezama, Kathy Yelick
NAS Benchmarks in Titanium

MG Line Count Comparison

- Fortran w/ MPI: 60
- Titanium: 552

CG Line Count Comparison

- Fortran w/ MPI: 20
- Titanium: 99

MG Class D Speedup - Opteron/IB

- Linear Speedup
- Fortran w/ MPI
- Titanium

CG Class D Speedup - G5/IB

- Linear Speedup
- Fortran w/ MPI
- Titanium

Work by Kaushik Datta, Dan Bonachea, Kathy Yelick
**High Level Optimizations in Titanium**

- Irregular communication can be expensive
  - “Best” strategy differs by data size/distribution and machine parameters
  - E.g., packing, sending bounding boxes, fine-grained updates

- Use of runtime optimizations
  - Inspector-executor

- Performance on Sparse MatVec Mult

- Results: best strategy differs within the machine on a single matrix (~ 50% better)

Average and maximum speedup of the Titanium version relative to the Aztec version on 1 to 16 processors

Work by Jimmy Su and Kathy Yelick
Titanium Compiler/Language Status

• Titanium runs on almost any machine
  • Requires a C compiler and C++ for the translator
  • Pthreads for shared memory
  • GASNet for distributed memory [Bonachea et al]
    • Shared with Berkeley UPC compiler

• Easily ported to future machines
• Base language mostly compatible with Java 1.4

• However, compiler/language no longer under development
Conclusions

• High performance programming need not be low level programming
• Java performance now rivals C
  • Look at industrial efforts for hints of the future
• Titanium adds key features for HPC
• Demonstrated effectiveness on real applications
  • Heart, cochlea, and soon ocean modeling (AMR)
• Research problems remain
  • Mixed parallelism model
  • Automatic communication optimizations
  • Performance portability
Outline

• Titanium Execution Model
• Titanium Memory Model
• Support for Serial Programming
• Performance and Applications
• **Hierarchical Execution Model**
  • Hierarchical teams
  • Dynamic replacement for single
  • Benchmark results
• **Hierarchical Memory Model**
Application Hierarchy

- Applications can reduce communication costs by adapting to machine hierarchy
  - Slow, avoid
  - Fast, allow

- Applications may also have inherent, algorithmic hierarchy
  - Recursive algorithms
  - Composition of multiple algorithms
  - Hierarchical division of data
Algorithm Example: Merge Sort

• Task parallel

```java
int[] mergeSort(int[] data) {
    int len = data.length;
    if (len < threshold)
        return sequentialSort(data);
    d1 = fork mergeSort(data[0:len/2-1]);
    d2 = mergeSort(data[len/2:len-1]);
    join d1;
    return merge(d1, d2);
}
```

• Cannot fork threads in SPMD
  • Must rewrite to execute over fixed set of threads
Algorithm Example: Merge Sort

- **SPMD**

```java
int[] mergeSort(int[] data, int[] ids) {
    int len = data.length;
    int threads = ids.length;
    if (threads == 1) return sequentialSort(data);
    if (myId in ids[0:threads/2-1])
        d1 = mergeSort(data[0:len/2-1], ids[0:threads/2-1]);
    else
        d2 = mergeSort(data[len/2:len-1], ids[threads/2:threads-1]);
    barrier(ids);
    if (myId == ids[0]) merge(d1, d2);
}
```
Thread Teams

• Thread teams are basic units of cooperation
  • Groups of threads that cooperatively execute code
  • Collective operations over teams

• Other languages have teams
  • MPI communicators, UPC teams

• However, those teams are flat
  • Do not match hierarchical structure of algorithms, machines
  • Misuse of teams can result in deadlock

```java
Team t1 = new Team(0:7);
Team t2 = new Team(0:3);
if (myId == 0) barrier(t1);
else barrier(t2);
```
Structured Teams

- Structured, hierarchical teams are the solution
  - Expressive: match structure of algorithms, machines
  - Safe: eliminate many sources of deadlock
  - Composable: enable clean composition of multiple algorithms or tasks
  - Efficient: allow users to take advantage of machine structure, resulting in performance gains
Team Data Structure

- Threads comprise teams in tree-like structure
- First-class object to allow easy creation and manipulation

![Team Data Structure Diagram]
**Machine Structure**

- Provide mechanism for querying machine structure and thread mapping at runtime

```java
Team T = Ti.defaultTeam();
```
Language Constructs

• Thread teams may execute distinct tasks
  \[
  \text{partition}(T) \left\{ \\
  \begin{array}{l}
  \{ \text{model\_fluid}() ; \} \\
  \{ \text{model\_muscles}() ; \} \\
  \{ \text{model\_electrical}() ; \} \\
  \end{array}
  \right. \\
  \]

• Threads may execute the same code on different sets of data as part of different teams
  \[
  \text{teamsplit}(T) \left\{ \\
  \begin{array}{l}
  \text{row\_reduce}() ; \\
  \end{array}
  \right. \\
  \]

• Scoping rules prevent some types of deadlock
  • Execution team determined by enclosing construct
Partition Semantics

- Different subteams of \( T \) execute each of the branches

```c
partition(T) {
    { model_fluid(); }
    { model_muscles(); }
    { model_electrical(); }
}
```
Teamsplit Semantics

- Each subteam of rowTeam executes the reduction on its own

```java
teamsplit(rowTeam) {
    Reduce.add(mttmp, myResults0, rpivot);
}
```
Multiple Hierarchy Levels

- Constructs can be nested
  ```
  teamsplit(T) {
    teamsplit(T.myChildTeam()) {
      level1_work();
    }
    level2_work();
  }
  ```

- Program can use multiple teams
  ```
  teamsplit(columnTeam) {
    myOut.vbroadcast(cpivot);
  }
  teamsplit(rowTeam) {
    Reduce.add(mtmp, myResults0, rpivot);
  }
  ```
Sorting

- Distributed sorting application using new hierarchical constructs

- Three pieces: sequential, shared memory, and distributed
  - Sequential quick sort from Java 1.4 library
  - Shared memory merge sort
  - Distributed memory sample sort
Shared Memory Hierarchy

- Team hierarchy is binary tree
- Trivial construction

```java
static void divideTeam(Team t) {
    if (t.size() > 1) {
        t.splitTeam(2);
        divideTeam(t.child(0));
        divideTeam(t.child(1));
    }
}
```

- Threads walk down to bottom of hierarchy, sort, then walk back up, merging along the way
SMP Sort and Merge Logic

- Control logic for sorting and merging

```java
static single void sortAndMerge(Team t) {
    if (Ti.numProcs() == 1) {
        allRes[myProc] = sequentialSort(myData);
    } else {
        teamsplit(t) {
            sortAndMerge(t.myChildTeam());
        }
        Ti.barrier();
        if (Ti.thisProc() == 0) {
            int otherProc = myProc + t.child(0).size();
            int[1d] myRes = allRes[myProc];
            int[1d] otherRes = allRes[otherProc];
            int[1d] newRes = target(t.depth(), myRes, otherRes);
            allRes[myProc] = merge(myRes, otherRes, newRes);
        }
    }
}
```
Algorithms for Hierarchical Machines

- Three strategies for hierarchical machines (e.g. clusters of SMPs):
  1. Treat the machine as a flat collection of processors that don’t share memory
  2. Compose a distributed communication library (e.g. MPI) with a shared memory library (e.g. Pthreads)
  3. Implement a hierarchical algorithm that takes advantage of both shared memory and all available concurrency

- Sort example:
  - Pure sample sort treats the machine as flat
  - Hierarchical sort uses sampling/distribution between shared-memory domains, SMP sort in a node
**Flat vs. Hierarchical Sort**

Distributed Sort (Cray XE6)
(10,000,000 elements/core, 10,000 samples/core)

- **Time (s)**

- **NUMA Nodes (6 cores/node)**

- **Bar Chart**:
  - **flat (distribution)**
  - **hierarchical (distribution)**
  - **flat (sort)**
  - **hierarchical (sort)**

**Good**
Communication Concurrency

Sort Communication Concurrency (Cray XE6)
(10,000,000 elements/core, 10,000 samples/core)

Distribution Time (s)

NUMA Nodes (6 cores/node)

- 1 thread/node
- 6 threads/node

Good
**Communication Concurrency**

Stencil Communication Concurrency
(256³ Points/Node, 100 Timesteps, Cray XE6)

- 1 thread/node
- 6 threads/node

- Total Time (s)
- NUMA Nodes (6 cores/node)

Good
Dynamic Alignment of Collectives

• Misaligned collective operations can result in deadlock
• Enforcing textual alignment of collectives at runtime can provide safety while minimizing programmer burden
• Basic idea:
  • Track control flow on all threads
  • Check that preceding control flow matches when:
    • Performing a team collective
    • Changing team contexts
• Compiler instruments source code to perform tracking and checking
### Checking Example

5 \textbf{if} (Ti\_thisProc() == 0)  
6 \hspace{1em} Ti\_barrier();  
7 \textbf{else}  
8 \hspace{1em} Ti\_barrier();

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0dc7637a</td>
<td>...*</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x0dc7637a</td>
<td>...*</td>
<td></td>
</tr>
</tbody>
</table>

* Entries prior to line 5
Checking Example

5 if (Ti.thisProc() == 0)
   6 Ti.barrier();
else
   8 Ti.barrier();

Control flow decision noted, hash updated

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td></td>
<td>...*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td></td>
<td>...*, (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5
Checking Example

5 if (Ti.thisProc() == 0)
0 Hash broadcast from thread 0
6 Ti.barrier();
7 else
8 Ti.barrier();

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td>0x7e8a6fa0</td>
<td>…*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td>…*, (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5
Checking Example

5 if (Ti.thisProc() == 0)
6 Ti.barrier();
7 else
8 Ti.barrier();

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td>0x7e8a6fa0</td>
<td>…*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td>…*, (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5

Hash from 0 compared with local hash
Checking Example

```java
5 if (Ti.thisProc() == 0)
6 Ti.barrier();
7 else
8 Ti.barrier();
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td>ERROR 3a6fa0</td>
<td>...*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td>...*, (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5
Checking Example

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>MISALIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td>ERROR 0x7e8a6fa0</td>
<td>...* (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td>...* (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5

```java
5 if (Ti.thisProc() == 0) {
  6 Ti.barrier();
} else {
  8 Ti.barrier();
}
```

Meaningful error generated
Minimal Overhead to Checking

Cluster Applications Time

Time Relative to Static

Processors

2 4 8 16 32

CG

FT

MG

Good
Conclusions

• Hierarchical language extensions simplify job of programmer
  • Can organize application around machine characteristics
  • Easier to specify algorithmic hierarchy
  • Seamless code composition
  • Better productivity, performance with team collectives
    • See paper for details

• Language extensions are safe to use
  • Safety provided by dynamic scoping and dynamic alignment checking
Outline

• Titanium Execution Model
• Titanium Memory Model
• Support for Serial Programming
• Performance and Applications
• Hierarchical Execution Model
• Hierarchical Memory Model
  • Pointer span
  • Hierarchical pointer analysis
  • Race detection results
Partitioned Global Address Space

- Partitioned global address space (PGAS) abstraction provides illusion of shared memory on non-shared memory machines
- Pointers can reference local or remote data
  - Location of data can be reflected in type system
  - Runtime handles any required communication

```c
double[1d] local srcl = new double[0:N-1];
double[1d] srcg = broadcast srcl from 0;
```
Hierarchical Memory

• PGAS model can be extended to hierarchical arrangement of memory spaces
• Pointers have varying span specifying how far away the referenced object can be
  • Reflect communication costs
**Pointer Span and Machine Team**

- Span of pointer related to level of least common ancestor of the source thread and the potential targets in the machine hierarchy.
  - \( \text{span} = \# \text{ of levels} - \text{target level} \)

<table>
<thead>
<tr>
<th>Span 4</th>
<th>Span 3</th>
<th>Span 2</th>
<th>Span 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
<td>0, 1, 2, 3</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>Level 0</td>
<td>Level 1</td>
<td>Level 2</td>
<td>Level 3</td>
</tr>
</tbody>
</table>

Diagram:
- Span 4: 0, 1, 2, 3, 4, 5, 6, 7
- Span 3: 0, 1, 2, 3, 4, 5, 6, 7
- Span 2: 0, 1, 2, 3, 4, 5, 6, 7
- Span 1: 0, 1, 2, 3, 4, 5, 6, 7

Arrows: 1, 2, 3, 4
**Pointers and Arbitrary Teams**

- Pointer span can be generalized to handle arbitrary teams
  - “Span” of pointer is now the combination of a specific team hierarchy and a level in that hierarchy
Pointers and Multiple Teams

• Relationship between teams can be represented as a lattice
• Span of a pointer is an element of the lattice
• Pointer analysis can determine span of pointers
Hierarchical Pointer Analysis

- Pointer analysis possible over hierarchical teams
  - Allocation sites $\rightarrow$ abstract locations (alocs)
  - Variables $\rightarrow$ points-to sets of alocs
- Abstract locations have span (e.g. thread local, global)
- SPMD model simplifies analysis
  - Allows effects of an operation on all threads to be simultaneously computed
  - Results are the same for all threads
**Pointer Analysis: Allocation**

- Allocation creates new thread local abstract location
  - Result of allocation must reside in local memory

```java
static void bar() {
    L1: Object b, a = new Object();
    teamsplit(t2) {
        b = broadcast a from 0;
    }
}
```

<table>
<thead>
<tr>
<th>Alocs</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>(1, thread local)</td>
</tr>
<tr>
<td>b</td>
<td>(1, (t2, 1))</td>
</tr>
</tbody>
</table>
**Pointer Analysis: Communication**

- Communication produces version of source abstract locations with greater span
  - Collective takes into account team over which it is executed

```java
static void bar() {
    Object b, a = new Object();
    teamsplit(t2) {
        b = broadcast a from 0;
    }
}
```

<table>
<thead>
<tr>
<th>Alocs</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>(1, thread local)</td>
</tr>
<tr>
<td>b</td>
<td>(1, (t2, 1))</td>
</tr>
</tbody>
</table>
Evaluation

• Pointer analysis implemented for 3-level machine hierarchy
• Evaluated on five application benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Line Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr</td>
<td>7581</td>
<td>Adaptive mesh refinement suite</td>
</tr>
<tr>
<td>gas</td>
<td>8841</td>
<td>Hyperbolic solver for a gas dynamics problem</td>
</tr>
<tr>
<td>cg</td>
<td>1595</td>
<td>NAS conjugate gradient benchmark</td>
</tr>
<tr>
<td>ft</td>
<td>1192</td>
<td>NAS Fourier transform benchmark</td>
</tr>
<tr>
<td>mg</td>
<td>1952</td>
<td>NAS multigrid benchmark</td>
</tr>
</tbody>
</table>
**Running Time**

- Determine cost of introducing hierarchy into pointer analysis
- Tests run on 2.93GHz Core i7 with 8GB RAM
- Three analysis variants compared

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA1</td>
<td>Single-level pointer analysis</td>
</tr>
<tr>
<td>PA2</td>
<td>Two-level pointer analysis (thread-local and global)</td>
</tr>
<tr>
<td>PA3</td>
<td>Three-level pointer analysis</td>
</tr>
</tbody>
</table>
Low Overhead for Hierarchy

Pointer Analysis Running Time

- amr
- gas
- cg
- ft
- mg

Benchmark

Time (s)

PA1 | PA2 | PA3

Good
Race Detection

- Pointer analysis used with concurrency analysis to detect potential races at compile-time
- Three analyses compared

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concur</td>
<td>Concurrency analysis plus constraint-based data sharing analysis and type-based alias analysis</td>
</tr>
<tr>
<td>concur+PA1</td>
<td>Concurrency analysis plus single-level pointer analysis</td>
</tr>
<tr>
<td>concur+PA3</td>
<td>Concurrency analysis plus three-level pointer analysis</td>
</tr>
</tbody>
</table>
More Precise Results

Static Race Detection

- Benchmark: amr, gas, cg, ft, mg
- Metrics: Possible Races (Log Scale)
- Comparisons: concur, concur+PA1, concur+PA3

Graph showing the number of possible races for each benchmark with logarithmic scale.

Benchmark | concur | concur+PA1 | concur+PA3
--- | --- | --- | ---
amr | 11493 | 505 | 12
| gas | 3067 | 45 | 5
| cg | 1474 | 25 | 12
| ft | 755 | 16 | 3
| mg | 4393 | 20 | 3

Evaluation: Good