# Global Address Space Programming in Titanium

**Kathy Yelick**

**CS267**



## Titanium Goals

- Performance
  - close to C/FORTRAN + MPI or better
- Safety
  - as safe as Java, extended to parallel framework
- Expressiveness
  - close to usability of threads
  - add minimal set of features
- Compatibility, interoperability, etc.
  - no gratuitous departures from Java standard

## Titanium

- **Take the best features of threads and MPI**
  - global address space like threads (ease programming)
  - SPMD parallelism like MPI (for performance)
  - local/global distinction, i.e., layout matters (for performance)
- **Based on Java, a cleaner C++**
  - classes, memory management
- **Language is extensible through classes**
  - domain-specific language extensions
  - current support for grid-based computations, including AMR
- **Optimizing compiler**
  - communication and memory optimizations
  - synchronization analysis
  - cache and other uniprocessor optimizations



## New Language Features

- **Scalable parallelism**
  - SPMD model of execution with global address space
- **Multidimensional arrays**
  - points and index sets as first-class values to simplify programs
  - iterators for performance
- **Checked Synchronization**
  - single-valued variables and globally executed methods
- **Global Communication Library**
- **Immutable classes**
  - user-definable non-reference types for performance
- **Operator overloading**
  - by demand from our user community
- **Semi-automated zone-based memory management**
  - as safe as a garbage-collected language
  - better parallel performance and scalability

## Lecture Outline

- Linguistic support for uniprocessor performance
  - Immutable classes
  - Multidimensional Arrays
  - foreach
- Parallelism Support
  - SPMD execution
  - Global and local references
  - Communication
  - Barriers and single
  - Synchronized (not yet implemented)
- Example: Sharks and Fish
- Java introduction interspersed
- Compiler status



## Java: A Cleaner C++

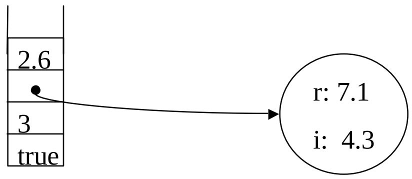
- Java is an object-oriented language
  - classes (no standalone functions) with methods
  - inheritance between classes; multiple interface inheritance only
- Documentation on web at java.sun.com
- Syntax similar to C++

```
class Hello {
   public static void main (String [] argv) {
           System.out.println(“Hello, world!”);
   }
}
```

- Safe
  - Strongly typed: checked at compile time, no unsafe casts
  - Automatic memory management
- Titanium is (almost) strict superset

# Java Objects

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






# 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

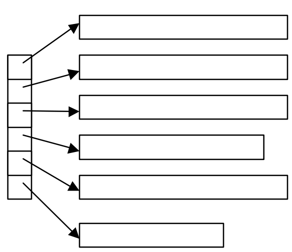
- For small objects, would sometimes prefer
  - to avoid level of indirection
  - pass by value (copying of entire object)
  - especially when objects are immutable -- fields are unchangeable
    - » extends the idea of primitive values (1, 4.2, etc.) to user-defined values
- Titanium introduces immutable classes
  - all fields are final (implicitly)
  - cannot inherit from (extend) or be inherited by other classes
  - needs to have 0-argument constructor, e.g., `Complex ()`

```
immutable class Complex { ... }
Complex c = new Complex(7.1, 4.3);
```



## Arrays in Java

- Arrays in Java are objects
- Only 1D arrays are directly supported
- Array bounds are checked
- Multidimensional arrays as arrays-of-arrays are slow



## Multidimensional Arrays in Titanium

- **New kind of multidimensional array added**
  - **Two arrays may overlap (unlike Java arrays)**
  - **Indexed by Points (tuple of ints)**
  - **Constructed over a set of Points, called Domains**
  - **RectDomains are special case of domains**
  - **Points, Domains and RectDomains are built-in immutable classes**
- **Support for adaptive meshes and other mesh/grid operations**

```
RectDomain<2> d = [0:n,0:n];
Point<2> p = [1, 2];
double [2d] a = new double [d];
a[0,0] = a[9,9];
```



## Naïve MatMul with Titanium Arrays

```
public static void matMul(double [2d] a, double [2d] b,
                          double [2d] c) {
  int n = c.domain().max()[1];  // assumes square
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
           c[i,j] += a[i,k] * b[k,j];
         }
      }
   }
}
```

## Unordered iteration

- As seen in matmul, we need to reorder iterations
- Compilers can (in principle) do this for matrix multiply, but hard in general
- Titanium adds unordered iteration on rectangular domains

```
foreach (p within r) { … }
```

  - p is a Point new point, scoped only within the foreach body
  - r is a previously-declared RectDomain
- Foreach simplifies bounds checking as well
  - note: current optimizer does not include bounds checks
- Additional operations on domains and arrays to subset and transform



## Better MatMul with Titanium Arrays

```
public static void matMul(double [2d] a, double [2d] b,
                          double [2d] c) {
  foreach (ij within c.domain()) {
    double [1d] aRowi = a.slice(1, ij[1]);
    double [1d] bColj = b.slice(2, ij[2]);
    foreach (k within aRowi.domain()) {
      c[ij] += aRowi[k] * bColj[k];
    }
  }
}
```

Note that code is still unblocked.

# Point, RectDomain, Arrays in General

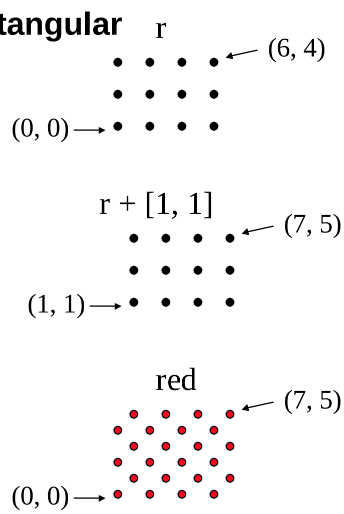
- Points specified by a tuple of ints
- RectDomains given by:
  - lower bound point
  - upper bound point
  - stride point
- Array given by RectDomain and element type

```
Point<2> lb = [1, 1];
Point<2> ub = [10, 20];
RectDomain<2> R = [lb : ub : [2, 2]];
double [2d] A = new double[r];
...
foreach (p in A.domain()) {
        A[p] = B[2 * p + [1, 1]];
}
```



# Example: Domain

- Domains in general are not rectangular
- Built using set operations
  - union, +
  - intersection, *
  - difference, -
- Example is red-black algorithm




```
Point<2> lb = [0, 0];
Point<2> ub = [6, 4];
RectDomain<2> r = [lb : ub : [2, 2]];
…
Domain<2> red = r + (r + [1, 1]);
foreach (p in red) {
        ...
}
```

## Example using Domains and foreach

- **Gauss-Seidel red-black computation in multigrid**

```
void gsrb() {
  boundary (phi);
  for (domain<2> d = res; d != null;
                        d = (d == red ? black : null)) {
    foreach (q in d)                    ⟸  unordered iteration
      res[q] = ((phi[n(q)] + phi[s(q)] + phi[e(q)] + phi[w(q)])*4
               + (phi[ne(q) + phi[nw(q)] + phi[se(q)] + phi[sw(q)])
               - 20.0*phi[q] - k*rhs[q]) * 0.05;
      foreach (q in d) phi[q] += res[q];
  }
}
```



## SPMD Execution Model

- **Java programs can be run as Titanium, but the result will be that all processors do all the work**
- **E.g., parallel hello world**

```
class HelloWorld {
public static void main (String [] argv) {
  System.out.println(“Hello from proc ”
                         Ti.thisProc());
}
}
```

- **Any non-trivial program will have communication and synchronization between processors**

## SPMD Execution Model

- A common style is compute/communicate
- E.g., in each timestep within fish simulation with gravitation attraction

```
read all fish and compute forces on mine
Ti.barrier();
write to my fish using new forces
Ti.barrier();
```



## SPMD Model

- All processor start together and execute same code, but not in lock-step
- Sometimes they take different branches

```
if (Ti.thisProc() == 0) { … do setup … }
for(all data I own) { … compute on data … }
```

- Common source of bugs is barriers or other global 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 the same value on all procs

```
int single timestep = 0;
```

# SPMD Execution Model

- Barriers and single in FishSimulation

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

- Single on methods may be inferred by compiler



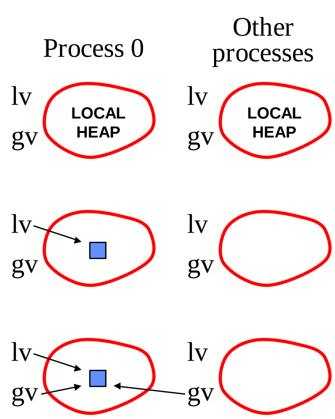
# Global Address Space

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

```
Class C { …int val;… }

C gv;       // global pointer
C local lv; // local pointer

if (thisProc() == 0) {
      lv = new C();
}
gv = broadcast lv from 0;
gv.val = …; // full
… = gv.val; // functionality
```



## Use of Global / Local

- **Default is global**
  - opposite of Split-C
  - easier to port shared-memory programs
  - harder to use sequential kernels
- **Use** local **declarations in critical sections**
  - same trade-off as Split-C
  - (same implementation as Split-C)
  - shared memory: no performance implications
  - distributed memory:
    - » save overhead of a few instructions when using a global reference to access a local object



## Distributed Data Structures

- **Build distributed data structures:**
  - broadcast or exchange

```
RectDomain <1> single allProcs = [0:Ti.numProcs-1];
RectDomain <1> myFishDomain = [0:myFishCount-1];
Fish [1d] single [1d] allFish =
                         new Fish [allProcs][1d];
Fish [1d] myFish = new Fish [myFishDomain];
allFish.exchage(myFish);
```

- **Now each processor has an array of global pointers, one to each processors chunk of fish**

## Consistency Model

- Titanium adopts the Java memory consistency model
- Roughly: Access to shared variables that are not synchronized have undefined behavior.
- Use synchronization to control access to shared variables.
  - barriers
  - synchronized methods and blocks



## Other Language Extensions

**Java extensions for expressiveness & performance**

- Operator overloading
- Zone-based memory management

**The following are not yet implemented in the compiler**

- Parameterized types (aka templates)
  - watching for standard
- Foreign function interface

## Implementation

- Strategy
  - compile Titanium into C
  - Solaris or Posix threads for SMPs
  - Active Messages (Split-C library) for communication
  - MPI (*)
- Status
  - runs on SUN Enterprise 8-way SMP
  - runs on Berkeley NOW
  - T3E port may be available by end of semester (*)
  - Clump port may be available by end of semester (*)
  - tuning for performance (*)
- (*) Indicates area for possible term projects



## Applications

- Three-D AMR Poisson Solver (AMR3D)
  - block-structured grids
  - 2000 line program
  - algorithm not yet fully implemented in other languages
  - tests performance and effectiveness of language features
- Other 2D Poisson Solvers (under development)
  - infinite domains
  - based on method of local corrections
- Three-D Electromagnetic Waves (EM3D)
  - unstructured grids
- Several smaller benchmarks

# Current Sequential Performance

- Taken on Ultrasparc
- Roughly 10x faster than JDK version of Java
- Compare codes written using Java arrays and Titanium arrays

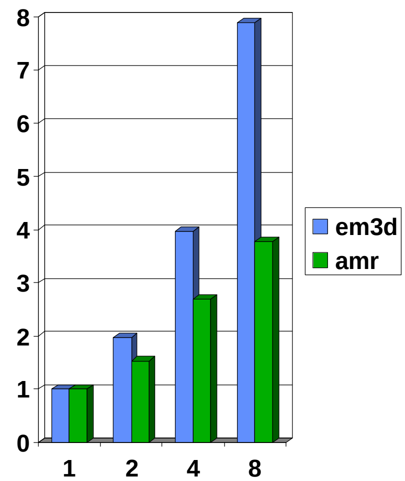
| | C/C++/ FORTRAN | Java Arrays | Titanium Arrays | Overhead |
|---|---|---|---|---|
| DAXPY | 1.4s | 6.8s | 1.5s | 7% |
| 3D multigrid | 12s | | 26s | 117% |
| 2D multigrid | 5.4s | | 6.2s | 15% |
| EM3D | 0.7s | 1.8s | 1.0s | 42% |

- More work to do here



# Parallel performance

- Speedup on Ultrasparc SMP
- AMR largely limited by
  - current algorithm
  - problem size
  - 2 levels, with top one serial
- Not yet optimized with "local" for distributed memory



## How to use Titanium

- Documentation on
  - http://www.cs.berkeley.edu/projects/titanium
  - Includes: Reference manual (terse), tutorial (incomplete), compiler documentation;
- To run compiler:
  - use path /disks/srs/titanium/sparc-sun-solaris2.6/bin/
  - use tcbuild Myprog.ti
    - » Myprog.ti is the titanium file containing class Myprog
    - » class Myprog has main method
    - » creates executable Myprog
  - tcbuild --backend smp-narrow for smp code
  - tcbuild --backend split-c for NOW code
  - tcbuild --help for more information
- Debugger also exist (sequential code only)



## Recommended Use

- If writing from scratch, may start by writing Java code (faster compiler, not faster code)
- Next use sequential Titanium
  - may omit data layout and problem partitioning
- Next use smp Titanium
  - need to partition work, but not data
- Finally, optimize for NOW
  - Any code the runs on an SMP should run correctly (if slowly) without modifications on the NOW.
  - Only exceptions:
    - » your code contains race conditions
    - » our compiler contains bugs (please report)

## Caveats

- Performance on the NOW is still being optimized (report egregious problems to us)
- Garbage collection does not work on NOW -- need to use regions
- Static has MPI-like meaning, not threads
  - one copy of a static per processor
- Bounds checking is not on by default



## Titanium Status

- Titanium language definition complete.
- Titanium compiler running.
- Compiles for uniprocessors, NOW; others soon.
- Application developments ongoing.
- Lots of research opportunities.