

Titanium: A Java Dialect for High Performance Computing

Dan Bonachea

U.C. Berkeley
and LBNL

<http://titanium.cs.berkeley.edu>

(slides courtesy of Kathy Yelick)

1

Titanium Group (Past and Present)

- Susan Graham
- Katherine Yelick
- Paul Hilfinger
- Phillip Colella (LBNL)
- Alex Aiken
- Ben Liblit
- Peter McQuorquodale (LBNL)
- Sabrina Merchant
- Carleton Miyamoto
- Chang Sun Lin
- Geoff Pike
- Luigi Semenzato (LBNL)
- Jimmy Su
- Tong Wen (LBNL)
- Siu Man Yau
- Arvind Krishnamurthy
- (and many undergrad researchers)

2

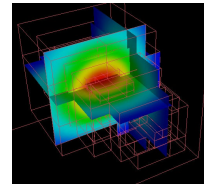
Motivation: Target Problems

- **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:**
 - Adaptive methods
 - Large scale parallel machines
- **Titanium is designed for methods with**
 - Structured grids
 - Locally-structured grids (AMR)
 - Unstructured grids (in progress)

3

Common Requirements

- **Algorithms for numerical PDE computations are**
 - communication intensive
 - memory intensive
- **AMR makes these harder**
 - more small messages
 - more complex data structures
 - most of the programming effort is debugging the boundary cases
 - locality and load balance trade-off is hard



4

Titanium

- Based on Java, a cleaner C++
 - classes, automatic memory management, etc.
 - compiled to C and then native binary (no JVM)
- Same parallelism model as UPC and CAF
 - SPMD with a global address space
 - Dynamic Java threads are not supported
- Optimizing compiler
 - static (compile-time) optimizer, not a JIT
 - communication and memory optimizations
 - synchronization analysis (e.g. static barrier analysis)
 - cache and other uniprocessor optimizations

5

Summary of Features Added to Java

- **Multidimensional arrays with iterators & copy ops**
- **Immutable (“value”) classes**
- **Templates**
- **Operator overloading**
- **Scalable SPMD parallelism**
- **Global address space**
- **Checked Synchronization**
- **Zone-based memory management (regions)**
- **Support for N-dim points, rectangles & point sets**
- **Libraries for collective communication, distributed arrays, bulk I/O, performance profiling**

6

Outline

- **Titanium Execution Model**
 - SPMD
 - Global Synchronization
 - Single
- **Titanium Memory Model**
- **Support for Serial Programming**
- **Performance and Applications**
- **Compiler/Language Status**
- **Compiler Optimizations & Future work**

7

SPMD Execution Model

- Titanium has the same execution model as UPC and CAF
- Basic Java programs may be run as Titanium, but 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

8

SPMD Model

- All processors start together and execute same code, but not in lock-step
- Basic control done using
 - `Ti.numProcs()` => total number of processors
 - `Ti.thisProc()` => id of executing processor
- Bulk-synchronous style
 - read all particles and compute forces on mine
 - `Ti.barrier();`
 - write to my particles using new forces
 - `Ti.barrier();`
- This is neither message passing nor data-parallel

9

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 to understanding compiler messages

10

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 processors will have the same value after the broadcast.

11

Example of Data Input

- Same example, but reading from keyboard
- Shows use of Java exceptions
 - int myCount = 0;
 - int single allCount = 0;
 - if (Ti.thisProc() == 0)
 - try {
 - DataInputStream kb = new
 - DataInputStream(System.in);
 - myCount =
 - Integer.valueOf(kb.readLine()).intValue();
 - } catch (Exception e) {
 - System.err.println("Illegal Input");
 - }
 - allCount = broadcast myCount from 0;

12

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 in such loops, if statements, etc. 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]

13

Single Variable Example

- **Barriers and single in N-body Simulation**

```
class ParticleSim {
public static void main (String [] argv) {
    int single allTimestep = 0;
    int single allEndTime = 100;
    for (; allTimestep < allEndTime; allTimestep++){
        read all particles and compute forces on mine
        Ti.barrier();
        write to my particles using new forces
        Ti.barrier();
    }
}
```
- **Single methods inferred by the compiler**

14

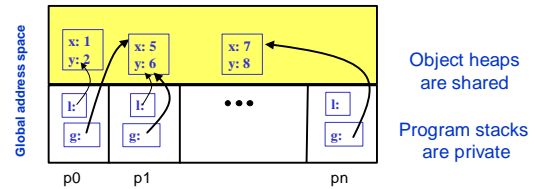
Outline

- **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**
- **Compiler/Language Status**
- **Compiler Optimizations & Future work**

15

Global Address Space

- Globally shared address space is partitioned
- References (pointers) are either local or global (meaning possibly remote)



16

Use of Global / Local

- **As seen, global references (pointers) may point to remote locations**
 - easy to port shared-memory programs
- **Global pointers are more expensive than local**
 - True even when data is on the same processor
 - Use `local` declarations in critical inner loops
- **Costs of global:**
 - space (processor number + memory address)
 - dereference time (check to see if local)
- **May declare references as local**
 - Compiler will automatically infer them when possible

17

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 (Ti.thisProc() == 0) {
    lv = new C();
}
gv = broadcast lv from 0;
gv.val = ...;
... = gv.val;
```



18

Shared/Private vs Global/Local

- Titanium's global address space is based on pointers rather than shared variables
- There is no distinction between a private and shared heap for storing objects
 - Although recent compiler analysis infers this distinction and uses it for performing optimizations [Liblit et. al 2003]
- All objects may be referenced by global pointers or by local ones
- There is no direct support for distributed arrays
 - Irregular problems do not map easily to distributed arrays, since each processor will own a set of objects (sub-grids)
 - For regular problems, Titanium uses pointer dereference instead of index calculation
 - Important to have local "views" of data structures

19

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.:

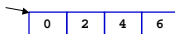

```
int [1d] arr;
arr = new int [1:100];
arr[1] = 4*arr[1];
```
- Will discuss these more later...

20

Explicit Communication: Exchange

- To create shared data structures
 - each processor builds its own piece
 - pieces are exchanged (for object, just exchange pointers)
- Exchange primitive in Titanium


```
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:

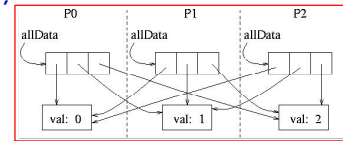


21

Building Distributed Structures

- Distributed structures are built with **exchange**:

```
class Boxed {
public Boxed (int j) { val = j;}
public int val;
}
```



```
Object [1d] single allData;
allData = new Object [0:Ti.numProcs()-1];
allData.exchange(new Boxed(Ti.thisProc()));
```

22

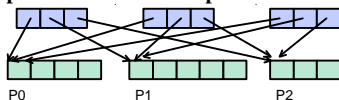
Distributed Data Structures

- 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);
```

All to all broadcast

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



23

Region-Based Memory Management

- An advantage of Java over C/C++ is:
 - Automatic memory management
- But unfortunately, garbage collection:
 - Has a reputation of slowing serial code
 - Is hard to implement and scale in a distributed environment
- Titanium takes the following approach:
 - Memory management is safe – cannot deallocate live data
 - Garbage collection is used by default (most platforms)
 - Higher performance is possible using region-based explicit memory management

24

Region-Based Memory Management

- Need to organize data structures
- Allocate set of objects (safely)
- Delete them with a single explicit call (fast)
 - David Gay's Ph.D. thesis

```
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");
}
}
```

25

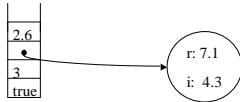
Outline

- Titanium Execution Model
- Titanium Memory Model
- Support for Serial Programming
 - Immutables
 - Operator overloading
 - Multidimensional arrays
 - Templates
- Performance and Applications
- Compiler/Language Status
- Compiler Optimizations & Future work

26

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 standard library
 - always allocated dynamically
 - passed by pointer value (object sharing) into functions
 - has level of indirection (pointer to) implicit
 - simple model, but inefficient for small objects



27

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 { ... }
```

28

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 datatypes
- 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
- Example uses:
 - Complex numbers, xyz components of a field vector at a grid cell (velocity, force)
- Note: considering lang. extension to allow mutation

29

Example of Immutable Classes

```
immutable class Complex {
    Complex () {real=0; imag=0; }
    ...
}
```

new keyword 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
 - Allows efficient support of complex types through a general language mechanism

30

Operator Overloading

- For convenience, Titanium provides operator overloading
 - important for readability in scientific code
 - Very similar to operator overloading in C++
 - Must be used judiciously

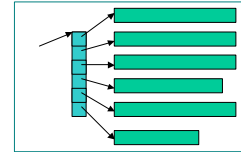
```
class Complex {
private double real;
private double imag;
public Complex op+(Complex c) {
    return new Complex(c.real + real,
                       c.imag + imag);
}
```

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

31

Arrays in Java

- Arrays in Java are objects
- Only 1D arrays are directly supported
- Multidimensional arrays are arrays of arrays
- General, but slow - due to memory layout, difficulty of compiler analysis, and bounds checking



- 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

32

Multidimensional Arrays in Titanium

- New multidimensional array added
 - One array may be a subarray of another
 - e.g., a is interior of b, or a is all even elements of b
 - can easily refer to rows, columns, slabs or boundary regions as sub-arrays of a larger array
 - Indexed by Points (tuples of ints)
 - Constructed over a rectangular set of Points, called Rectangular Domains (RectDomains)
 - Points, Domains and RectDomains are built-in immutable classes, with handy literal syntax
- Expressive, flexible and fast
- Support for AMR and other grid computations
 - domain operations: intersection, shrink, border
 - bounds-checking can be disabled after debugging phase

33

Unordered Iteration

- Memory hierarchy optimizations are essential
 - Compilers can sometimes do these, but hard in general
 - Titanium adds explicitly unordered iteration over domains
 - Helps the compiler with loop & dependency analysis
 - Simplifies bounds-checking
 - Also avoids some indexing details - more concise
- ```
foreach (p in r) { ... A[p] ... }
```
- p is a Point (tuple of ints) that can be used to index arrays
  - r is a RectDomain or Domain
- Additional operations on domains to subset and xform
  - Note: foreach is not a parallelism construct

34

## Point, RectDomain, Arrays in General

- Points specified by a tuple of ints
 

```
Point<2> lb = [1, 1];
Point<2> ub = [10, 20];
```
- RectDomains given by 3 points:
  - lower bound, upper bound (and optional stride)

```
RectDomain<2> r = [lb : ub];
```
- Array declared by num dimensions and type
 

```
double [2d] a;
```
- Array created by passing RectDomain
 

```
a = new double [r];
```

35

## Simple Array Example

- Matrix sum in Titanium

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

No array allocation here

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

Syntactic sugar

Optional stride

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

Equivalent loops

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

36

## 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];
 }
 }
 }
}
```

37

## Better MatMul with Titanium Arrays

```
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];
 }
 }
}
```

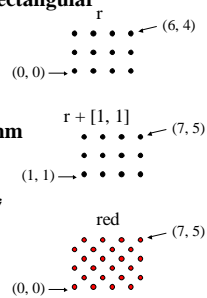
Current performance: comparable to 3 nested loops in C  
Recent upgrades: automatic blocking for memory hierarchy (Geoff Pike's PhD thesis)

38

## 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) {
 ...
}
```



39

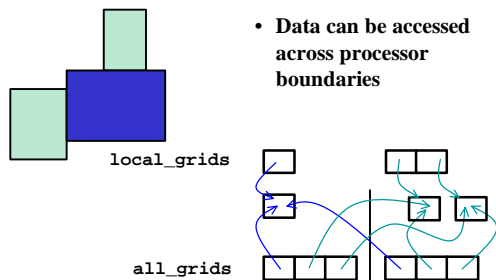
## Example using Domains and foreach

- Gauss-Seidel red-black computation in multigrid

```
void gsrb() {
 boundary(phi);
 for (Domain<2> d = red; 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)]) * 4
 - k * rhs[q]) * 0.05;
 foreach (q in d) phi[q] += res[q];
 }
}
```

40

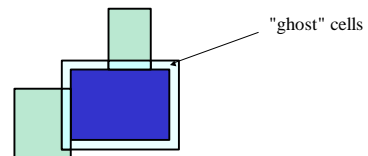
## Example: A Distributed Data Structure



41

## Example: Setting Boundary Conditions

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



42

## Templates

- **Many applications use containers:**
  - E.g., arrays parameterized by dimensions, element types
  - Java supports this kind of parameterization through inheritance
    - Can only put Object types into containers
    - Inefficient when used extensively
- **Titanium provides a template mechanism closer to that of C++**
  - E.g. Can be instantiated with "double" or immutable class
  - Used to build a distributed array package
  - Hides the details of exchange, indirection within the data structure, etc.

43

## Example of Templates

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

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

- Addresses programmability and performance

44

## 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;
 }
}

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

45

## 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
- **Compiler/Language Status**
- **Compiler Optimizations & Future work**

46

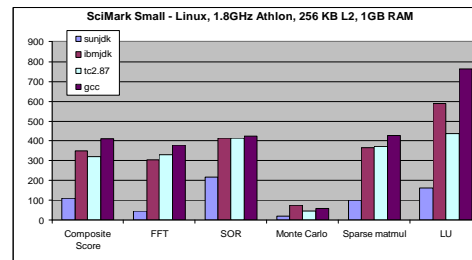
## SciMark Benchmark

- **Numerical benchmark for Java, C/C++**
  - purely sequential
- **Five kernels:**
  - FFT (complex, 1D)
  - Successive Over-Relaxation (SOR)
  - Monte Carlo integration (MC)
  - Sparse matrix multiply
  - dense LU factorization
- **Results are reported in MFlops**
  - We ran them through Titanium as 100% pure Java with no extensions
- **Download and run on your machine from:**
  - <http://math.nist.gov/scimark2>
  - C and Java sources are provided

Roldan Pozo, NIST, <http://math.nist.gov/~Rpozo>

47

## Java Compiled by Titanium Compiler

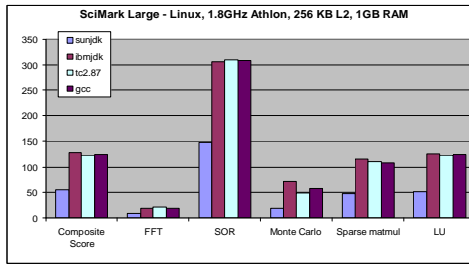


- Sun JDK 1.4.1\_01 (HotSpot(TM) Client VM) for Linux
- IBM J2SE 1.4.0 (Classic VM cxia32140-20020917a, jite JIT) for 32-bit Linux
- Titaniumc v2.87 for Linux, gcc 3.2 as backend compiler -O3, no bounds check
- gcc 3.2, -O3 (ANSI-C version of the SciMark2 benchmark)

48



## 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, jitc 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)

49

## Sequential Performance of Java

- **State of the art JVM's**
  - often very competitive with C performance
  - within 25% in worst case, sometimes better than C
- **Titanium compiling pure Java**
  - On par with best JVM's and C performance
  - This is without leveraging Titanium's lang. extensions
- **We can try to do even better using a traditional compilation model**
  - Berkeley Titanium compiler:
    - Compiles Java + extensions into C
    - No JVM, no dynamic class loading, whole program compilation
    - Do not currently optimize Java array accesses (prototype)

50

## Language Support for Performance

- **Multidimensional arrays**
  - Contiguous storage
  - Support for sub-array operations without copying
- **Support for small objects**
  - E.g., complex numbers
  - Called "immutables" in Titanium
  - Sometimes called "value" classes
- **Unordered loop construct**
  - Programmer specifies loop iterations independent
  - Eliminates need for dependence analysis (short term solution?) Same idea used by vectorizing compilers.

51

## Array Performance Issues

- **Array representation is fast, but access methods can be slow, e.g., bounds checking, strides**
- **Compiler optimizes these**
  - common subexpression elimination
  - eliminate (or hoist) bounds checking
  - strength reduce: e.g., naive code has 1 divide per dimension for each array access
- **Currently +/- 20% of C/Fortran for large loops**
- **Future: small loop and cache tiling optimizations**

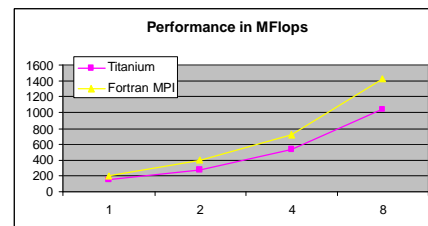
52

## Applications in Titanium

- **Benchmarks and Kernels**
  - Fluid solvers with Adaptive Mesh Refinement (AMR)
  - Scalable Poisson solver for infinite domains
  - Conjugate Gradient
  - 3D Multigrid
  - Unstructured mesh kernel: EM3D
  - Dense linear algebra: LU, MatMul
  - Tree-structured n-body code
  - Finite element benchmark
  - SciMark serial benchmarks
- **Larger applications**
  - Heart and Cochlea simulation
  - Genetics: micro-array selection
  - Ocean modeling with AMR (in progress)

53

## NAS MG in Titanium

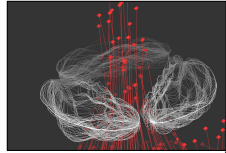


- **Preliminary Performance for MG code on IBM SP**
  - Speedups are nearly identical
  - About 25% serial performance difference

54

## Heart Simulation - Immersed Boundary Method

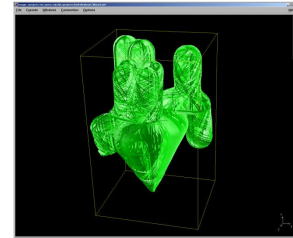
- Problem: compute blood flow in the heart
  - Modeled as an elastic structure in an incompressible fluid.
  - The “immersed boundary method” [Peskin and McQueen].
  - 20 years of development in model
  - Many other applications: blood clotting, inner ear, paper making, embryo growth, and more
- Can be used for design prosthetics
  - Artificial heart valves
  - Cochlear implants



55

## Simulating Fluid Flow in Biological Systems

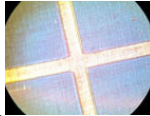
- Immersed Boundary Method
  - Material (e.g., heart muscles, cochlea structure) modeled by grid of material points
  - Fluid space modeled by a regular lattice
- Irregular material points need to interact with regular fluid lattice
  - Trade-off between load balancing of fibers and minimizing communication
  - Memory and communication intensive
  - Includes a Navier-Stokes solver and a 3-D FFT solver
- Heart simulation is complete, Cochlea simulation is close to done
  - First time that immersed boundary simulation has been done on distributed-memory machines
  - Working on a Ti library for doing other immersed boundary simulations



56

## MOOSE Application

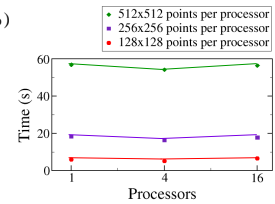
- Problem: Genome Microarray construction
  - Used for genetic experiments
  - Possible medical applications long-term
- Microarray Optimal Oligo Selection Engine (MOOSE)
  - A parallel engine for selecting the best oligonucleotide sequences for genetic microarray testing from a sequenced genome (based on uniqueness and various structural and chemical properties)
  - First parallel implementation for solving this problem
  - Uses dynamic load balancing within Titanium
  - Significant memory and I/O demands for larger genomes



57

## Scalable Parallel Poisson Solver

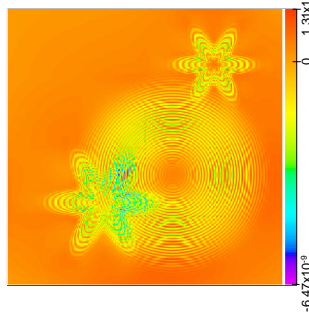
- MLC for Finite-Differences by Balls and Colella
- Poisson equation with infinite boundaries
  - arise in astrophysics, some biological systems, etc.
- Method is scalable
  - Low communication (<5%)
- Performance on
  - SP2 (shown) and T3E
  - scaled speedups
  - nearly ideal (flat)
- Currently 2D and non-adaptive



58

## Error on High-Wavenumber Problem

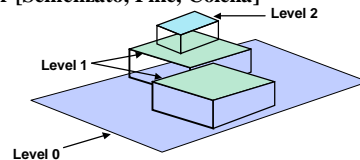
- Charge is
  - 1 charge of concentric waves
  - 2 star-shaped charges.
- Largest error is where the charge is changing rapidly.
  - Note:
    - discretization error
    - faint decomposition error
- Run on 16 procs



59

## AMR Poisson

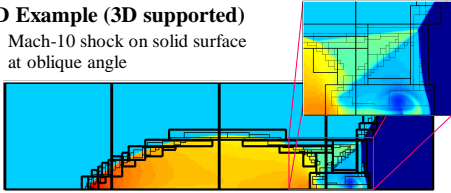
- Poisson Solver [Semenzato, Pike, Colella]
  - 3D AMR
  - finite domain
  - variable coefficients
  - multigrid across levels
- Performance of Titanium implementation
  - Sequential multigrid performance +/- 20% of Fortran
  - On fixed, well-balanced problem of 8 patches, each 72<sup>3</sup>
  - parallel speedups of 5.5 on 8 processors



60

## AMR Gas Dynamics

- **Hyperbolic Solver [McCorquodale and Colella]**
  - Implementation of Berger-Colella algorithm
  - Mesh generation algorithm included
- **2D Example (3D supported)**
  - Mach-10 shock on solid surface at oblique angle



- **Future: Self-gravitating gas dynamics package**

61

## Outline

- **Titanium Execution Model**
- **Titanium Memory Model**
- **Support for Serial Programming**
- **Performance and Applications**
- **Compiler/Language Status**
- **Compiler Optimizations & Future work**

62

## Titanium Compiler Status

- **Titanium compiler runs on almost any machine**
  - Requires a C compiler (and decent C++ to compile translator)
  - Pthreads for shared memory
  - Communication layer for distributed memory (or hybrid)
    - Recently moved to live on GASNet: shared with UPC
    - Obtained Myrinet, Quadrics, and improved LAPI implementation
- **Recent language extensions**
  - Indexed array copy (scatter/gather style)
  - Non-blocking array copy under development
- **Compiler optimizations**
  - Cache optimizations, for loop optimizations
  - Communication optimizations for overlap, pipelining, and scatter/gather under development

63

## Implementation Portability Status

- **Titanium has been tested on:**
    - POSIX-compliant workstations & SMPs
    - Clusters of uniprocessors or SMPs
    - Cray T3E
    - IBM SP
    - SGI Origin 2000
    - Compaq AlphaServer
    - MS Windows/GNU Cygwin
    - and others...
  - **Supports many communication layers**
    - High performance networking layers:
      - IBM/LAPI, Myrinet/GM, Quadrics/Elan, Cray/shmem, Infiniband (soon)
    - Portable communication layers:
      - MPI-1.1, TCP/IP (UDP)
- Automatic portability:  
Titanium applications run on all of these!  
Very important productivity feature for debugging & development
- <http://titanium.cs.berkeley.edu>

64

## Programmability

- Heart simulation developed in ~1 year
  - Extended to support 2D structures for Cochlea model in ~1 month
- **Preliminary code length measures**
  - Simple torus model
    - Serial Fortran torus code is 17045 lines long (2/3 comments)
    - Parallel Titanium torus version is 3057 lines long.
  - Full heart model
    - Shared memory Fortran heart code is 8187 lines long
    - Parallel Titanium version is 4249 lines long.
  - Need to be analyzed more carefully, but not a significant overhead for distributed memory parallelism

65

## Robustness

- **Robustness is the primary motivation for language “safety” in Java**
  - Type-safe, array bounds checked, auto memory management
  - Study on C++ vs. Java from Phipps at Spirus:
    - C++ has 2-3x more bugs per line than Java
    - Java had 30-200% more lines of code per minute
- **Extended in Titanium**
  - Checked synchronization avoids barrier/collective deadlocks
  - More abstract array indexing, retains bounds checking
- **No attempt to quantify benefit of safety for Titanium yet**
  - Would like to measure speed of error detection (compile time, runtime exceptions, etc.)
  - Anecdotal evidence suggests the language safety features are very useful in application debugging and development

66

## Calling Other Languages

- **We have built interfaces to**
  - PETSc : scientific library for finite element applications
  - Metis: graph partitioning library
  - KeLP: scientific C++ library
- **Two issues with cross-language calls**
  - accessing Titanium data structures (arrays) from C
    - possible because Titanium arrays have same format on inside
  - having a common message layer
    - Titanium is built on lightweight communication

67

## Outline

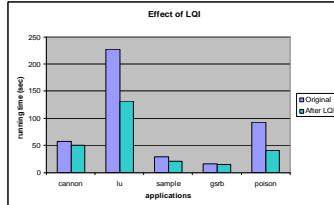
- **Titanium Execution Model**
- **Titanium Memory Model**
- **Support for Serial Programming**
- **Performance and Applications**
- **Compiler/Language Status**
- **Compiler Optimizations & Future work**
  - Local pointer identification (LQI)
  - Communication optimizations
  - Feedback-directed search-based optimizations

68

## Local Pointer Analysis

- **Global pointer access is more expensive than local**
- **Compiler analysis can frequently infer that a given global pointer always points locally**
  - Replace global pointer with a local one
  - **Local Qualification Inference (LQI)** [Liblit]
  - **Data structures must be well partitioned**

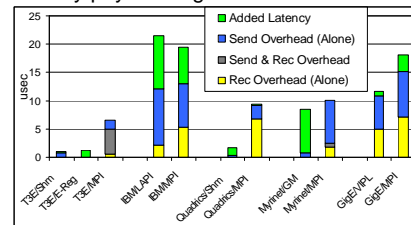
Same idea can be applied to UPC's pointer-to-shared...



69

## Communication Optimizations

- Possible communication optimizations
- Communication overlap, aggregation, caching
- Effectiveness varies by machine
- Generally pays to target low-level network API



[Bell, Bonachea et al] at IPDPS'03

70

## Split-C Experience: Latency Overlap

- **Titanium borrowed ideas from Split-C**
  - global address space
  - SPMD parallelism
- **But, Split-C had explicit non-blocking accesses built in to tolerate network latency on remote read/write**

```

int *global p;
x := *p; /* get */
p := 3; / put */
sync; /* wait for my puts/gets */

```
- **Also one-way communication**

```

p := x; / store */
all_store_sync; /* wait globally */

```
- **Conclusion: useful, but complicated**

71

## Titanium: 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
- **Open question: Can we leverage the relaxed consistency model to automate communication overlap optimizations?**
  - difficulty of alias analysis is a significant problem

72

## Sources of Memory/Comm. Overlap

- **Would like compiler to introduce put/get/store**
- **Hardware also reorders**
  - out-of-order execution
  - write buffered with read by-pass
  - non-FIFO write buffers
  - weak memory models in general
- **Software already reorders too**
  - register allocation
  - any code motion
- **System provides enforcement primitives**
  - e.g., memory fence, volatile, etc.
  - tend to be heavyweight and have unpredictable performance
- **Open question: Can the compiler hide all this?**

73

## Feedback-directed search-based optimization

- Use machines, not humans for architecture-specific tuning
  - Code generation + search-based selection
    - Can adapt to cache size, # registers, network buffering
  - Used in
    - Signal processing: FFTW, SPIRAL, UHFFT
    - Dense linear algebra: Atlas, PHI-PAC
    - Sparse linear algebra: Sparsity
    - **Rectangular grid-based computations: Titanium compiler**
      - Cache tiling optimizations - automated search for best tiling parameters for a given architecture

74

## Current Work & Future Plans

- Unified communication layer with UPC: GASNet
- Exploring communication overlap optimizations
  - Explicit (programmer-controlled) and automated
  - Optimize regular and irregular communication patterns
- Analysis and refinement of cache optimizations
  - along with other sequential optimization improvements
- Additional language support for unstructured grids
  - arrays over general domains, with multiple values per grid point
- Continued work on existing and new applications

<http://titanium.cs.berkeley.edu>

75