Titanium and Java Parallelism

Arvind Krishnamurthy
Fall 2004

Titanium

- Take the best features of threads and MPI (just like Split-C)
  - 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
  - support for grid-based computations, including adaptive mesh refinement (AMR)
- Optimizing compiler
  - compiled down to C
  - communication and memory optimizations
  - cache and other uniprocessor optimizations
Java: A Cleaner C++

- Java is an object-oriented language
  - classes (no standalone functions) with methods
  - inheritance between classes; multiple interface inheritance only
- Syntax similar to C++
  ```java
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
- 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

```
2.6
3
true
```

```
r: 7.1
i: 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
  - 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
  - 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
- Points specified by a tuple of ints
- RectDomains given by: lower bound, upper bound [stride]
- Array declared by # dimensions and type, created by passing domain

```java
Point<2> lb = [1, 1];
Point<2> ub = [10, 20];
RectDomain<2> r = [lb : ub];
double [2d] a = new double [r];
```
Unordered iteration

- Reordering iterations helps perform
- Compilers can (in principle) do this, but hard in general
- Titanium adds unordered iteration on rectangular domains
  
  ```java
  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
- Additional operations on domains and arrays to subset and transform

MatMul with Titanium Arrays

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

*Note that code is unblocked.*
Example: Domain

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

```plaintext
Point<2> lb = [0, 0];
Point<2> ub = [6, 4];
Point<2> s = [1, 1];
RectDomain<2> r = [lb : ub : [2, 2]];
RectDomain<2> r1 = [lb+s : ub+s : [2, 2]];
Domain<2> red = r + r1;
foreach (p in red) {
  ...
}
```

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
```java
class HelloWorld {
  public static void main (String [] argv) {
    System.out.println("Hello from proc "+
         Ti.thisProc());
  }
}
```
- Barrier synchronization: `Ti.barrier()`
**Safe Barriers**

- All processor start together and execute same code, but not in lock-step
- Sometimes they take different branches
  ```java
  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
  ```java
  barrier, broadcast, reduction, exchange
  ```
- A “single” method is one called by all procs
  ```java
  public single static void allStep(...)
  ```
- A “single” variable has the same value on all procs
  ```java
  int single timestep = 0;
  ```

**SPMD Execution Model**

- Barriers and single in FishSimulation
  ```java
  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

```c
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 performance 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
Memory Management

- Garbage collection
  - Reference counting
  - Copying garbage collection, generational garbage collection, etc.

- Distributed GC
  - Complex
  - Potentially expensive

- Zone-based memory management
  - extends existing model
  - good performance
  - safe
  - easy to use

Zone-Based Memory Management

- Allocate objects in zones
- Release zones manually

```java
Zone Z1 = new Zone();
Zone Z2 = new Zone();
T x = new(Z1) T();
T y = new(Z2) T();
x.field = y;
x = y;
delete Z1;
delete Z2;  // error
```
Consistency Model

- Titanium adopts the Java memory consistency model
  - Not sequential consistency
- 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

Sequential Consistency Recap

- Sequential consistency: parallel execution is a simple interleaving
- Guarantee: all previous operations are completed when a subsequent instruction is issued

```
P1 P2
/*Assume initial values of A and B are 0 */
(1a) A = 1;                    (2a) print B;
(1b) B = 2;                    (2b) print A;
```

- Valid outputs: [B = 2, A = 1], [B = 0, A = 1], [B = 0, A = 0]
- Invalid: [B = 2, A = 0]
- Sequential consistency could be violated by:
  - Compiler reordering the stores
  - Processor reordering the stores
  - Or if stores are nonblocking, get reordered by the network
  - Remember that blocking stores are expensive!
Reordering is sometimes valid...

Possible outputs: \([A = 0, B = 0], [A = 1, B = 0], [A = 0, B = 2], [A = 1, B = 2]\)

Since all combinations are valid, the stores can be reordered and one can't tell them apart from a un-reordered execution

<table>
<thead>
<tr>
<th>Actual Execution</th>
<th>Equivalent Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B = 2;)</td>
<td>print (A \Rightarrow 0)</td>
</tr>
<tr>
<td>print (A \Rightarrow 0)</td>
<td>(A = 1;)</td>
</tr>
<tr>
<td>print (B \Rightarrow 2)</td>
<td>(B = 2;)</td>
</tr>
<tr>
<td>(A = 1;)</td>
<td>print (B \Rightarrow 2)</td>
</tr>
</tbody>
</table>

Multiprocessor Implications

- Moral: OK to cheat if no one is looking!
- Depends on the behavior of other threads
- On a multiprocessor, optimize by replacing blocking with non-blocking operations

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>/*Assume initial values of A and B are 0 */</td>
<td>/*Assume initial values of A and B are 0 */</td>
</tr>
<tr>
<td>(1a) (A := 1;)</td>
<td>(2a) print (A;)</td>
</tr>
<tr>
<td>(1b) (B := 2;)</td>
<td>(2b) print (B;)</td>
</tr>
<tr>
<td>(lc) (\text{sync};)</td>
<td></td>
</tr>
</tbody>
</table>

- Other useful optimizations:
  - “caching” \(\Rightarrow\) fetch a value once, store it in a local variable, and use it multiple times
  - Extends to common subexpression elimination
Java Consistency Model

- Compiler/runtime-system/architecture free to reorder loads and stores of variables

- If strict ordering is required:
  - Use “volatile” keyword for variables (A and B in previous example will be declared as volatile)
  - Or use “synchronized” – Java’s locking construct

```java
synchronized (this) {
    A = 1;
    B = 2;
}
```
  - Obtain and release lock on “this”
  - When lock is released, guarantee that all previous operations are complete

Interaction with aliasing

- Java’s consistency model is stronger than coherence but weaker than sequential consistency
- Programmer needs to be aware of this, and program with caution
- Java’s goal was to enable compiler optimizations, but it turns out that there are subtle interactions with alias analysis

```
x ← read A  x ← read A
y ← read B  y ← read B
z ← read A  z ← x
```

Only if A and B are not aliased.

- Study of Java consistency model is still an open problem
Other Features & Summary

- Templates
  - Parameterized class declarations
  - For ease of programming
- “Exchange”
  - Concurrent broadcast of one value from each processor
  - Gather all of the broadcasted values into an array on all processors
- Compiler notes:
  - Way-ahead-of-time (WAT) compiler
  - Source-to-source translator
- Summary: Java-based global address language
  - Safe, cleaner (than C)
  - Wins back most of the performance degradation of pure Java