Making Sequential Consistency Practical in Titanium

Amir Kamil and Jimmy Su
Definition: A parallel execution must behave as if it were an interleaving of the serial executions by individual threads, with each individual execution sequence preserving the program order.

Initially, $\text{flag} = \text{data} = 0$

### Legal execution: $a \times y \ b$

### Illegal execution: $x \ y \ b \ a$

Critical cycle
Motivation

• Reduce the cost of sequential consistency in Titanium programs
  – Fences are inserted for memory accesses that can run concurrently to specify order
  – Inserted fences can prevent optimizations such as code motion and communication aggregation

• In order to reduce the number of fences, precisely find all pairs of heap accesses to the same location that can run concurrently
Titanium Features

• *Barrier*: the thread executing the barrier waits until all other threads have executed the same *textual* instance of the barrier call.
  – Example:
    
    `work1(); Ti.barrier(); work2();`

• A *single* value expression has the same value on all threads.
  – Example:
    
    `Ti.numProcs() == 2`
  – For a branch guarded by a single value expression, all threads are guaranteed to take the same branch.
Concurrency Analysis (I)

- Graph generated from program as follows:
  - Node added for each code segment between barriers and single conditionals
  - Edges added to represent control flow between segments

// code segment 1
if ([single])
    // code segment 2
else
    // code segment 3
// code segment 4
Ti.barrier()
// code segment 5
Concurrency Analysis (II)

- Two accesses can run concurrently if:
  - They are in the same node, or
  - One access’s node is reachable from the other access’s node without hitting a barrier

- Algorithm: remove barrier edges, do DFS

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
```
Thread-Aware Alias Analysis

• Two types of abstract locations: local and remote

• Remote locations created on demand when necessary
  – points-to set of remote location is remote analog of points-to set of corresponding local location

• Two locations $A$ and $B$ may alias across threads if:
  $\exists \ x \in \text{pointsTo}(A). \ \text{R}(x) \in \text{pointsTo}(B)$,
  (where $\text{R}(x)$ is the remote counterpart of $x$)
Thread-Aware AA Example

class phase20 {
    public static void main(String[] args) {
        L1:   phase20 a = new phase20();
             phase20 b = broadcast a from 0;
        L2:   a.z = new Object();
        L3:   b.z = new Object();
    }
    L4: Object z = new Object();
}

<table>
<thead>
<tr>
<th>Points-to Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>a  $\rightarrow{1}$</td>
</tr>
<tr>
<td>b  $\rightarrow{1,1_r}$</td>
</tr>
<tr>
<td>1.z $\rightarrow{4,2,3,3_r}$</td>
</tr>
<tr>
<td>1_r.z $\rightarrow{4_r,2_r,3_r,3}$</td>
</tr>
</tbody>
</table>
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines(^1)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>56</td>
<td>Monte Carlo integration</td>
</tr>
<tr>
<td>demv</td>
<td>122</td>
<td>Dense matrix-vector multiply</td>
</tr>
<tr>
<td>sample-sort</td>
<td>321</td>
<td>Parallel sort</td>
</tr>
<tr>
<td>lu-fact</td>
<td>420</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>3d-fft</td>
<td>614</td>
<td>Fourier transform</td>
</tr>
<tr>
<td>gsrb</td>
<td>1090</td>
<td>Computational fluid dynamics kernel</td>
</tr>
<tr>
<td>gsrb(^*)</td>
<td>1099</td>
<td>Slightly modified version of gsrb</td>
</tr>
<tr>
<td>spmv</td>
<td>1493</td>
<td>Sparse matrix-vector multiply</td>
</tr>
<tr>
<td>gas</td>
<td>8841</td>
<td>Hyperbolic solver for gas dynamics</td>
</tr>
</tbody>
</table>

\(^1\) Line counts do not include the reachable portion of the 37,000 line Titanium/Java 1.0 libraries
Fence Counts

Percentages are for number of dynamic fences reduced over naive

<table>
<thead>
<tr>
<th>naive</th>
<th>All heap accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>sharing</td>
<td>All shared accesses</td>
</tr>
<tr>
<td>concur/taa/cycle</td>
<td>Concurrency analysis + thread-aware AA + cycle detection</td>
</tr>
</tbody>
</table>
Optimizations

- Overlap bulk memory copies
- Communication aggregation for irregular array accesses (ie $a[b[i]]$)
- Both optimizations reorder accesses, so sequential consistency can prevent them
**Performance Results**

**Sparse Matrix Vector Multiply**
- Bar chart showing speedup for different numbers of processors (1, 2, 4, 8, 16) for relaxed, naive, sharing, and concur/taa/cycle strategies.

**Dense Matrix Vector Multiply**
- Bar chart showing speedup for different numbers of processors (1, 2, 4, 8, 16) for relaxed, naive, sharing, and concur/taa/cycle strategies.

Linux cluster with Itanium/Myrinet

**Conclusion:** sequential consistency can be provided with little or no performance cost.