Concurrency Analysis for Parallel Programs with Textually Aligned Barriers

Amir Kamil and Katherine Yelick
Titanium Group
http://titanium.cs.berkeley.edu

U.C. Berkeley
October 20, 2005
Motivation/Goals

• Many program analyses and optimizations benefit from knowing which expressions can run concurrently
• Develop basic concurrency analysis for Titanium programs
• Refine analysis to ignore *infeasible* paths
• Evaluate analysis using two applications
  • Race detection
  • Memory model enforcement
**Barrier Alignment**

- Many parallel languages make no attempt to ensure that barriers line up
  - Example code that is legal but will deadlock:
    ```java
    if (Ti.thisProc() % 2 == 0)
        Ti.barrier();  // even ID threads
    else
        ;  // odd ID threads
    ```
Aiken and Gay introduced *structural correctness* (*POPL’98*)

- Ensures that every thread executes the same number of barriers
- Example of structurally correct code:

```java
if (Ti.thisProc() % 2 == 0)
    Ti.barrier(); // even ID threads
else
    Ti.barrier(); // odd ID threads
```
Textual Barrier Alignment

- Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers
  - Stronger guarantee than structural correctness – this example is illegal:
    ```java
    if (Ti.thisProc() % 2 == 0)
    Ti.barrier(); // even ID threads
    else
    Ti.barrier(); // odd ID threads
    ```

- *Single-valued* expressions used to enforce textual barriers
Single-Valued Expressions

• A single-valued expression has the same value on all threads when evaluated
  • Example: Ti.numProcs() > 1

• All threads guaranteed to take the same branch of a conditional guarded by a single-valued expression
  • Only single-valued conditionals may have barriers
  • Example of legal barrier use:
    
    ```
    if (Ti.numProcs() > 1)
        Ti.barrier(); // multiple threads
    else
        ; // only one thread
    ```
Concurrency Graph

• Represents concurrency as a graph
  • Nodes are program expressions
  • If a path exists between two nodes, they can run concurrently

• Generated from control flow graph

```
x++;  
Ti.barrier();  
if (b) z--;  
else z++;  
if (c [single]) w--;  
else y++;  
foo();
```
Barriers

• Barriers prevent code before and after from running concurrently
• Nodes for barrier expressions removed from concurrency graph

```java
x++;  
Ti.barrier();  
...  
```

![Diagram of barriers removing nodes from concurrency graph](image)
Non-Single Conditionals

• Branches of a non-single conditional can run concurrently
  • Different threads can take different branches

• Edge added in the concurrency graph between branches

```java
if (b)
    z--;  
else
    z++;  
...  
```

[Diagram showing concurrency graph with edges between branches]
Single Conditionals

• Branches of a single conditional cannot run concurrently
  • All threads take the same branch
• No edge added in the concurrency graph between branches

```c
if (c [single])
    w--;
else
    y++;  
...  
```

```mermaid
graph TD
    c --> w--
    c --> y++
    w-- --> ...
    y++ --> ...
```

10 Amir Kamil
**Method Calls**

- Method call nodes split into a call and return node
  - Edges added from call node to target method’s subgraph, and from target method to return node

```java
... foo();
```
Concurrency Algorithm

• Two accesses can run concurrently if at least one is reachable from the other
• Concurrent accesses computed by doing $N$ depth first searches

```plaintext
x++;  
Ti.barrier();  
if (b) z--;  
else z++;  
if (c [single]) w--;  
else y++;  
foo();
```
**Infeasible Paths (I)**

• Handling of method calls allows *infeasible* control flow paths
  
  • Path exists into one call site and out of another

```
call foo
method bar
ret foo
```

```
call foo
method foo
ret foo
```

```
call foo
method baz
ret foo
```
Infeasible Paths (II)

- Solution – label call and return edges with matching parentheses
  - Only follow paths that correspond to balanced parentheses
Bypass Edges (I)

• Reachability now depends on context
• Inefficient to revisit method in every context
• Solution – add edges to bypass method calls
**Bypass Edges (II)**

- Can only bypass method calls that can actually complete (without executing a barrier)
- Iteratively compute set of methods that can complete

\[
\text{CanComplete} \leftarrow \varphi \\
\text{Do (until a fixed point is reached):} \\
\text{CanComplete} \leftarrow \text{CanComplete} \cup \text{all methods that can complete by only calling methods in } \text{CanComplete}
\]
Static Race Detection

• Two heap accesses compose a data race if they can concurrently access the same location, and at least one is a write.

Initially, \( x = 0 \)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Possible final values of ( x ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>set ( x = x + 1 )</td>
<td>set ( x = x - 1 )</td>
<td>-1, 0, 1</td>
</tr>
</tbody>
</table>

• Alias and concurrency analysis used to statically compute set of possible data races:
  • Analyses are sound, so all real races are detected.
• Goal is to minimize number of false races detected.
Sequential Consistency

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 [Lamport79].

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

Legal execution: $a \ x \ y \ b$

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

Critical cycle

Titanium and most other languages do not provide sequential consistency due to the (perceived) cost of enforcing it.
Enforcing Sequential Consistency

• Compiler and architecture must not reorder memory accesses that are part of a critical cycle

• Fences inserted into program to enforce order
  • Potentially costly – can prevent optimizations such as code motion and communication aggregation
  • At runtime, can cost an RTT on a distributed machine

• Goal is to minimize number of inserted fences
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines(^1)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lu-fact</td>
<td>420</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>gsrβ</td>
<td>1090</td>
<td>Computational fluid dynamics kernel</td>
</tr>
<tr>
<td>spmv</td>
<td>1493</td>
<td>Sparse matrix-vector multiply</td>
</tr>
<tr>
<td>pps</td>
<td>3673</td>
<td>Parallel Poisson equation solver</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
**Analysis Levels**

- We tested analyses of varying levels of precision

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>All expressions assumed concurrent</td>
</tr>
<tr>
<td>concur</td>
<td>Basic concurrency analysis</td>
</tr>
<tr>
<td>feasible</td>
<td>Feasible paths concurrency analysis</td>
</tr>
</tbody>
</table>

- All levels use alias analysis to distinguish memory locations
Race Detection Results

Number of Data Races Detected

Benchmark:
- gas
- gsr
- lu-fact
- pps
- spmv

Fraction Compared to base

Legend:
- base
- concur
- feasible

LCPC ’05: Concurrency Analysis 22 Amir Kamil
Sequential Consistency Results

Number of Static Memory Barriers

Fraction Compared to base

Benchmark

gas  gsrb  lu-fact  pps  spmv

base  concur  feasible

Sequential Consistency Results
Conclusion

• Textual barriers and single-valued expressions allow for simple but precise concurrency analysis

• Concurrency analysis is useful both for detecting races and for enforcing sequential consistency
  • Not sufficient for race detection – too many false positives
  • Good enough for sequential consistency to be provided at low cost (SC|05)

• Ignoring infeasible paths significantly improves analysis results