MAMA: Mostly Automatic Management of Atomicity

Christian DeLozier, Joseph Devietti, Milo M. K. Martin
University of Pennsylvania

March 2nd, 2014
Start with a serial problem
Find and express the parallelism
Coordinate the parallel execution (synchronization)
Don’t mess up!
Is there another way to do this?

• Programmer currently has to:
  1. Express the parallelism (Hard)
  2. Coordinate the parallelism (Hard)

• Alternative:
  1. Programmer expresses the parallelism
  2. Machine handles coordination
Coordinating Parallel Execution

- Atomicity vs. Ordering
  - Types of concurrency bugs [Lu et al., ASPLOS 2008]
  - Atomicity: Locks, transactions
  - Ordering: Barriers, fork/join, blocking on a queue, etc.

- Atomicity constraints are more common than ordering constraints

- Difficult to infer ordering constraints
Mostly Automatic Management of Atomicity

- Toward automatically providing atomicity for parallel programs
- Program either executes atomically—or deadlocks
- Protect every shared variable with its own lock
- Restore progress and performance when necessary (with help from the programmer)

Christian DeLozier - 2014
Related Work

• Automatic Parallelization
  • [Bernstein, IEEE Transactions 1966]
  • …

• Data Centric Synchronization
  • [Vaziri et. al, POPL 2006]
  • [Ceze et. al, HPCA 2007]

• Transactional Memory
  • [Herlihy and Moss, ISCA 1993]
  • …
Lock-Based Atomic Sections

• What lock do we acquire?

• When do we acquire the lock?

• When should we release the lock?
What lock do we acquire?

- Associate a lock with each variable

- Trade-off between parallelism and overhead

- Coarse-grained vs. Fine-grained
  - Coarse-grained: 1 lock per object, 1 lock per array
  - Fine-grained: 1 lock per field, 1 lock per array element

- Mutex vs. Reader-writer lock
MAMA Prototype

• Uses fine-grained locking
  • More parallelism
    • Especially for arrays
  • Optimization: Divide arrays into N chunks, 1 lock per chunk

• Uses reader-writer locks
  • More parallelism
    • Read sharing is common
Lock-Based Atomic Sections

• What lock do we acquire?
  • One reader-writer lock per variable (fine-grained)

• When do we acquire the lock?
  • Acquire before the first dynamic access

• When should we release the lock?
When should we release the lock?

- Simple case: After the owning thread has exited
When should we release the lock?

- When the owning thread is waiting for another thread to make progress (e.g. join, barrier)
When should we release the lock?

- Other deadlocks cannot be safely broken
- Need help from the programmer
  - Trusted annotations to sanction breaking a deadlock
    - \texttt{MAMA\_release(object)}
  - Also used to improve performance when threads are over-serialized
Lock-Based Atomic Sections

• What lock do we acquire?
  ■ One reader-writer lock per variable (fine-grained)

• When do we acquire the lock?
  ■ Acquire before the first dynamic access

• When should we release the lock?
  ■ At thread exit
  ■ When waiting for another thread to make progress
  ■ Or, at programmer sanctioned program points
What can deadlocks tell us?

• When a thread cannot acquire a lock:
  
  • Perform distributed deadlock detection
    [Bracha and Toueg, Distributed Computing 1987]

```c
void f()
{
    A = 1;
    B = 2;
}

void g()
{
    B = 1;
    A = 2;
}
```

T1

T2

Write B

Write A
MAMA Prototype

- Implemented as a RoadRunner tool [Flanagan and Freund, PASTE 2010]
  - Dynamic instrumentation for Java byte-code

- Evaluated on the Java Grande benchmarks and selected DaCapo benchmarks
  - Running on one socket (8 cores) of a 4 socket Nehalem system with 128 GB RAM

- Removed all synchronized blocks and java.util.concurrent constructs from benchmarks
  - Ensure that MAMA is providing all of the atomicity
Evaluating MAMA

• Can we execute parallel programs correctly?

• How many annotations need to be added for progress and performance?

• How is the performance of the program affected?
  • Does MAMA permit thread to execute in parallel?
## Annotation Burden

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Progress Annotations</th>
<th>Performance Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>314</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lufact</td>
<td>461</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>lusearch</td>
<td>124105</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>matmult</td>
<td>187</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>487</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>1165</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>pmd</td>
<td>60062</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>series</td>
<td>180</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sor</td>
<td>186</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sunflow</td>
<td>21970</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>xalan</td>
<td>172300</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
MAMA incurs overhead due to locking and serial execution. But, MAMA still allows some parallel execution as compared to serialization.
• Many benchmarks have significant portions that run in parallel
• Checking whether or not a lock is already owned incurs significant overhead on some benchmarks
• Fine-grained locking incurs significant memory overheads
• Could be optimized to save space via chunking arrays or decreasing the size of the lock
Future Directions

- Does this approach apply to other languages?

- How do we test programs running with MAMA?
  - Find uncommon deadlocks
  - Gain more confidence in trusted annotations

- How can we reduce the performance overheads?

- How can we infer ordering constraints?
MAMA

• Provides atomicity for parallel programs
  • Some help via annotations from programmer

• A step toward programming without worrying about atomicity
  • Programmer expresses parallelism
  • Machine provides atomicity automatically
Thank you for listening!