Problem

• Have a program
• Want it to run faster
Solution

• Profile program
• Find loops that take most time
• Parallelize each loop, run parallel program
  – No synchronization between parallel iterations
  – Measure quality of service, performance
  – Fix any quality of service, performance problems
    • Introduce synchronization
    • Privatize data
    • Replicate data
  – Use memory profiling, goal-directed approach
Quality of Service

• Number that measures relative difference between output from serial and parallel programs
• Smaller is better; zero indicates serial and parallel programs produce same output
• Quality of service of 0.10 indicates that parallel program produces output that differs from serial output by 10%
• We can run this basic approach without the quality of service concept – it will work even if we require the parallel program to produce the identical output as the serial program
Example

- Water computation (Jade benchmark suite)
- Simulates liquid water molecules ($N^2$ algorithm)
Execution Time Profiling

• Most of time spent in two loops: interf (63%), poteng (36%)

• Parallelize interf loop, run program (8 threads):

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Quality of Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.501</td>
<td>0.385</td>
</tr>
<tr>
<td>1.515</td>
<td>0.383</td>
</tr>
<tr>
<td>1.522</td>
<td>0.382</td>
</tr>
<tr>
<td>1.518</td>
<td>0.387</td>
</tr>
<tr>
<td>1.515</td>
<td>0.399</td>
</tr>
<tr>
<td>1.505</td>
<td>0.393</td>
</tr>
<tr>
<td>1.514</td>
<td>0.391</td>
</tr>
<tr>
<td>1.509</td>
<td>0.379</td>
</tr>
</tbody>
</table>
What Is Going Wrong?

• Potential quality of service problems (that we can do something about)
  – Atomicity violations
  – Interference on data that should be thread local
  – Data dependence violations

• Potential performance problems (that we can do something about)
  – True sharing between threads
  – False sharing between threads
  – Locality problems
How To Fix Problems?

• Get memory profiling information
  – Run program, generate trace containing
  – Loop iteration entry and exit events
  – Memory access events
    • Accessed address, type of access (read, write)
    • Instruction performing the access
• Consider each problem in turn
  – Come up with a potential fix for that problem
    (guided by memory profiling information)
  – Try the fix and see if it works!
## Sample Trace

<table>
<thead>
<tr>
<th>iter 0</th>
<th>iter 1</th>
<th>iter 2</th>
<th>iter 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k : rd(a₁)</td>
<td>k : rd(a₃)</td>
<td>k : rd(a₁)</td>
<td>k : rd(a₂)</td>
</tr>
<tr>
<td>l : wr(a₁)</td>
<td>l : wr(a₃)</td>
<td>l : wr(a₁)</td>
<td>l : wr(a₂)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k : rd(a₂)</td>
<td>k : rd(a₄)</td>
<td>k : rd(a₃)</td>
<td>k : rd(a₄)</td>
</tr>
<tr>
<td>l : wr(a₂)</td>
<td>l : wr(a₄)</td>
<td>l : wr(a₃)</td>
<td>l : wr(a₄)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i : rd(b)</td>
<td>i : rd(b)</td>
<td>i : rd(b)</td>
<td>i : rd(b)</td>
</tr>
<tr>
<td>j : wr(b)</td>
<td>j : wr(b)</td>
<td>j : wr(b)</td>
<td>j : wr(b)</td>
</tr>
</tbody>
</table>
Sample Trace Structure

<table>
<thead>
<tr>
<th>iter 0</th>
<th>iter 1</th>
<th>iter 3</th>
<th>iter 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k : rd(a₁)</td>
<td>k : rd(a₃)</td>
<td>k : rd(a₁)</td>
<td>k : rd(a₂)</td>
</tr>
<tr>
<td>l : wr(a₁)</td>
<td>l : wr(a₃)</td>
<td>l : wr(a₁)</td>
<td>l : wr(a₂)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k : rd(a₂)</td>
<td>k : rd(a₄)</td>
<td>k : rd(a₃)</td>
<td>k : rd(a₄)</td>
</tr>
<tr>
<td>l : wr(a₂)</td>
<td>l : wr(a₄)</td>
<td>l : wr(a₃)</td>
<td>l : wr(a₄)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i : rd(b)</td>
<td>i : rd(b)</td>
<td>i : rd(b)</td>
<td>i : rd(b)</td>
</tr>
<tr>
<td>j : wr(b)</td>
<td>j : wr(b)</td>
<td>j : wr(b)</td>
<td>j : wr(b)</td>
</tr>
</tbody>
</table>

2 Patterns

\{ k:rd(\alpha), l:wr(\alpha) \}

(pattern (8 instances))

\{ i:rd(\beta), j:wr(\beta) \}

(pattern (4 instances))
Synchronization Insertion for Atomicity

• Look at memory trace for interf loop
• See lots of patterns like κ : rd(α); λ : wr(α);
• Hypothesis is that lack of atomicity is problem
• Fix is to insert lock operations
  – κ : rd(α); λ : wr(α); becomes
  – acquire(α); κ : rd(α); λ : wr(α); release(α);
Prioritizing Synchronization Insertion

- Multiple patterns in trace
- Synchronize patterns one at a time (until result is acceptable)
- Prioritize according to interference density

\[
\text{Priority} = \frac{\sum \text{Number of Instructions that Interfere with that Instance}}{\text{All Instances of Pattern}}
\]

\[
\sum \text{Number of Instances of Pattern}
\]
# Sample Trace Structure

<table>
<thead>
<tr>
<th>iter 0</th>
<th>iter 1</th>
<th>iter 3</th>
<th>iter 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **iter 0**:
  - \( k : \text{rd}(a_1) \)
  - \( l : \text{wr}(a_1) \)
- **iter 1**: (highlighted)
  - \( k : \text{rd}(a_3) \)
  - \( l : \text{wr}(a_3) \)
- **iter 3**: (highlighted)
  - \( k : \text{rd}(a_1) \)
  - \( l : \text{wr}(a_1) \)
- **iter 4**: (highlighted)
  - \( k : \text{rd}(a_2) \)
  - \( l : \text{wr}(a_2) \)

## 2 Patterns

- **(8 instances)**
  - **Priority = 1**
    - \( k : \text{rd}(\alpha), \ l : \text{wr}(\alpha) \)
    - **Pattern**
    - **Priority = 1**

- **(4 instances)**
  - **Priority = 3**
    - \( i : \text{rd}(\beta), \ j : \text{wr}(\beta) \)
    - **Pattern**
    - **Priority = 3**
Synchronization Results

• Synchronize \texttt{i:rd(\beta), j:wr(\beta)} pattern first

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Quality of Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.491</td>
<td>0.00065</td>
</tr>
<tr>
<td>0.457</td>
<td>0.00048</td>
</tr>
<tr>
<td>0.455</td>
<td>0.00080</td>
</tr>
<tr>
<td>0.493</td>
<td>0.00038</td>
</tr>
<tr>
<td>0.446</td>
<td>0.00085</td>
</tr>
<tr>
<td>0.492</td>
<td>0.00044</td>
</tr>
<tr>
<td>0.494</td>
<td>0.00063</td>
</tr>
<tr>
<td>0.491</td>
<td>0.00030</td>
</tr>
</tbody>
</table>

• Quality of service good, performance bad
Next Hypothesis

• Performance is bad because of true sharing
• Replicate shared data
• Transform
  – acquire(\(\alpha\)); \(\kappa : \text{rd}(\alpha)\); \(\lambda : \text{wr}(\alpha)\); release(\(\alpha\));
  – \(t = \text{lookup}(\alpha)\); \(\kappa : \text{rd}(t)\); \(\lambda : \text{wr}(t)\);
• \textbf{lookup} accesses local replica of address \(\alpha\)
  – Hash table maps global addresses to local replicas
  – Initialize each replica to zero when initially accessed
  – Add replicas together at end of loop
  – Store sum in global address
Results After Replication

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Quality of Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.78</td>
<td>0.0044</td>
</tr>
<tr>
<td>1.77</td>
<td>0.0041</td>
</tr>
<tr>
<td>1.78</td>
<td>0.0034</td>
</tr>
<tr>
<td>1.80</td>
<td>0.0045</td>
</tr>
<tr>
<td>1.79</td>
<td>0.0040</td>
</tr>
<tr>
<td>1.81</td>
<td>0.0043</td>
</tr>
<tr>
<td>1.80</td>
<td>0.0043</td>
</tr>
<tr>
<td>1.77</td>
<td>0.0047</td>
</tr>
</tbody>
</table>

- Good performance, good quality of service
- So keep these transformations
- Move on to next loop (poteng)
## Parallelization of poteng Loop

<table>
<thead>
<tr>
<th>Parallelize poteng loop</th>
<th>Synchronize three ( \kappa : \text{rd}(\alpha); \lambda : \text{wr}(\alpha); ) patterns</th>
<th>Replicate three ( \kappa : \text{rd}(\alpha); \lambda : \text{wr}(\alpha); ) patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speedup</strong></td>
<td><strong>Quality of Service</strong></td>
<td><strong>Speedup</strong></td>
</tr>
<tr>
<td>2.35</td>
<td>0.645</td>
<td>0.698</td>
</tr>
<tr>
<td>2.29</td>
<td>0.641</td>
<td>0.703</td>
</tr>
<tr>
<td>2.38</td>
<td>0.649</td>
<td>0.695</td>
</tr>
<tr>
<td>2.35</td>
<td>0.645</td>
<td>0.691</td>
</tr>
<tr>
<td>2.43</td>
<td>0.655</td>
<td>0.708</td>
</tr>
<tr>
<td>2.38</td>
<td>0.649</td>
<td>0.690</td>
</tr>
<tr>
<td>2.32</td>
<td>0.647</td>
<td>0.691</td>
</tr>
<tr>
<td>2.36</td>
<td>0.649</td>
<td>0.688</td>
</tr>
</tbody>
</table>