Deterministic OpenMP

Amitai Aviram
Dissertation Defense
Department of Computer Science
Yale University
20 September 2012
Committee

- Bryan Ford, Yale University, Advisor
- Zhong Shao, Yale University
- Ramakrishna Gummadi, Yale University
- Emery Berger, University of Massachusetts-Amherst
The Big Picture

- OpenMP is a well-established annotation language to parallelize source code
- Deterministic OpenMP (DOMP) is our new version of OpenMP
  - Guarantees the same results for the same input
  - Enforces a deterministic programming model
  - Catches concurrency bugs
Unordered Memory Accesses

\[
\begin{align*}
  x &= 1 & x &= 2 \\
  x &= 1 & x &= 2 \quad ? \quad x &= 0 \\
  x &= 2 & y &= x \quad x &= 2 & y &= 0 \\
  x &= 2 & y &= 2 \quad ? \quad x &= 0 \\
  lock & & lock \\
  x++ & & x^* = 2 \\
  unlock & & unlock \\
  x &= 1 \quad ? \quad x &= 1 \\
  x &= 2 \quad ? \quad x &= 2 
\end{align*}
\]
A
lock(x)
x := x + 2
unlock(x)

B
lock(x)
x := x * 3
unlock(x)

But I got the right answer on 1,000,000 test runs ...

That's 'cause B happened to go first — until now!

Accesses remained unordered

HEISENBUG
Determinism

- *program*: input $\rightarrow$ (output, behavior)
- Results are as if memory accesses are always ordered
- Bugs are always reproducible
- Reproduce computations exactly
  - Byzantine fault tolerance
  - Accountability systems
  - Addressing timing channel attacks
Two Approaches

Run any parallel program deterministically, even a racy one.

Impose a deterministic schedule on the program.
Two Approaches

Run any parallel program deterministically, even a racy one.

Impose a deterministic schedule on the program.

Run only deterministic programs.

Enforce a deterministic programming model.
Two Approaches

Run any parallel program deterministically, even a racy one.
Impose a deterministic schedule on the program.

Run only deterministic programs.
Enforce a deterministic programming model.

Potentially useful but can be problematic
Two Approaches

Run any parallel program deterministically, even a racy one.
Impose a deterministic schedule on the program.

Run only deterministic programs.
Enforce a deterministic programming model.

Potentially useful but can be problematic

Determinator OS (OSDI '10)
Two Approaches

- Run any parallel program deterministically, even a racy one.
  - Impose a deterministic schedule on the program.

- Run only deterministic programs.
  - Enforce a deterministic programming model.

Potentially useful but can be problematic.

Determinator OS (OSDI '10)

DOMP
DOMP Semantics

- Based on familiar OpenMP API
- Excludes nondeterministic OpenMP constructs (*critical, atomic, flush*)
- Extends OpenMP: generalized reduction construct
- Implements a strict deterministic programming model
But can programmers really use a deterministic programming model?
Our Analysis

• Analyzed standard parallel benchmarks
• Counted instances of synchronization constructs
  • Deterministic (fork, join, barrier)
  • Nondeterministic (mutex locks, condition variables, etc.)
• Classified nondeterministic instances by use (idiom)
We found …

Programmers usually (74%) use nondeterministic primitives to build *deterministic* higher-level *idioms* for which the language lacks direct expression.
Making Determinism Accessible

- OpenMP API
- User library for Linux
- Replacement for GCC's OpenMP support library (libgomp)
- Often a drop-in replacement for libgomp
Making Determinism Accessible

- OpenMP API
- User library for Linux
- Replacement for GCC's OpenMP support library (libgomp)
- Often a drop-in replacement for libgomp

OUR GOAL
Outline

• The Big Picture ✓
• Background
• Analysis
• Design and Semantics
• Implementation
• Evaluation
• Conclusion
Outline

- The Big Picture ✓
- Background
- Analysis
- Design and Semantics
- Implementation
- Evaluation
- Conclusion
Single-Assignment Languages

- Dataflow languages
- Parallel Haskell
- Concurrency Collections (CnC)
Single-Assignment Languages

- Dataflow languages
- Parallel Haskell
- Concurrency Collections (CnC)

No data races
Single-Assignment Languages

- Dataflow languages
- Data Parallel Haskell
- Concurrency Collections (CnC)

No data races
Deterministic
Single-Assignment Languages

- Dataflow languages
- Data Parallel Haskell
- Concurrency Collections

- No data races
- Deterministic

UNFAMILIAR
Single-Assignment Languages

- Dataflow languages
- Data Parallel Haskell
- Concurrency Collections

No data races
Deterministic

UNFAMILIAR
Rewrite legacy code
Deterministic Imperative Languages

- SHIM
  - Message passing
- Deterministic Parallel Java (DPJ)
  - Programmer annotates data with effect classes
Deterministic Imperative Languages

- SHIM
  - Message passing
- Deterministic Parallel Java (DPJ)
  - Programmer annotates data with effect classes
Record-and-Replay Systems

- Instant Replay (1987)
- Recap (1988)
- DejaVu (1998)
- ReVirt (2002)
- Many others
Record-and-Replay Systems

- Instant Replay (1987)
- Recap (1988)
- DejaVu (1998)
- ReVirt (2002)
- Many others

SLOW
Record-and-Replay Systems

- Instant Replay (1987)
- Recap (1988)
- DejaVu (1998)
- ReVirt (2002)
- Many others

- OR -

SLOW

Require special $$$
Deterministic Schedulers

- DMP
- CoreDet
- Grace
- Dthreads
- Kendo
  - Orders lock acquisitions only
  - Racy programs remain nondeterministic
- Tern
  - Memoizes and re-uses schedules
Dedeterministic Scheduling

Thread A
- non-conflicting accesses
- conflicting accesses

Thread B
- parallel
- sequential
- parallel

...
Schedule Dependency

\[ x = 42; \]

// Thread A:
{
    if (input_is_typical)
        do_a_lot();
    x++; 
}

// Thread B:
{
    do_a_little();
    x++; 
}"
x = 42;
// Thread A:
{
    if (input_is_typical)
        do_a_lot();
    x++;
}
// Thread B:
{
    do_a_little();
    x++;
}
Schedule Dependency

\[
x = 42;
// Thread A:
\{
    if (input_is_typical)
        do_a_lot();
x++;
\}
// Thread B:
\{
    do_a_little();
x++;
\}
\]

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 \leftarrow \text{input_is_typical} )</td>
<td>call do_a_little</td>
</tr>
<tr>
<td>jump_zero ( t_1 ), L1</td>
<td>ret</td>
</tr>
<tr>
<td>call do_a_lot</td>
<td>( t_2 \leftarrow x )</td>
</tr>
<tr>
<td>ret</td>
<td>add ( t_2 ), 1</td>
</tr>
<tr>
<td>L1: ( t_1 \leftarrow x )</td>
<td></td>
</tr>
<tr>
<td>add ( t_1 ), 1</td>
<td></td>
</tr>
<tr>
<td>( x \leftarrow t_1 )</td>
<td>x \leftarrow t_2</td>
</tr>
</tbody>
</table>
Determinator OS
Determinator OS

Deterministic programming model
Limited API
Unconventional OS
Deterministic OpenMP (DOMP)

- Familiar, expressive OpenMP API
- Includes almost all constructs
- Excludes nondeterministic constructs
  - atomic, critical, flush
- Extends OpenMP with generalized reduction
- Enforces deterministic parallel programming model (like Determinator)
- User library for Linux
- Works with GCC
Outline

- The Big Picture ✓
- Background ✓
- Analysis
- Design and Semantics
- Implementation
- Evaluation
- Conclusion
Outline

- The Big Picture ✓
- Background ✓
- Analysis
- Design and Semantics
- Implementation
- Evaluation
- Conclusion
How easily could real programs conform to DOMP's deterministic programming model?
Method

- Used three parallel benchmark suites
  - SPLASH2, NPB-OMP, PARSEC
  - Total 35 benchmarks
- Hand-counted *instances* of synchronization constructs
- Recorded instances of deterministic constructs
- *Classified* and recorded instances of *nondeterministic* constructs by their *use*
Deterministic Constructs

- Fork/join
- Barrier
- OpenMP work sharing constructs
  - Loop
  - Master
  - (Sections)
  - (Task)
Nondeterministic Constructs

- Mutex lock/unlock
- Condition variable wait/broadcast
- (Semaphore wait/post)
- OpenMP *critical*
- OpenMP *atomic*
- (OpenMP *flush*)
Use in Idioms

```c
long ProcessId;

/* Get unique ProcessId */
LOCK(Global->CountLock);
    ProcessId = Global->current_id++;
UNLOCK(Global->CountLock);
```

barnes (SPLASH2)

Work sharing
Idioms

- Work sharing
- Reduction
- Pipeline
- Task queue
- Legacy
  - Obsolete: Making I/O or heap allocation thread safe
- Nondeterministic
  - Load balancing, random simulated interaction …
Work Sharing

“Data Parallelism”
cf. OpenMP LOOP work sharing construct

“Task Parallelism”
cf. OpenMP sections and task work sharing constructs
Reduction

X * ((((((X * V_0) * V_1) * V_2) * V_3) * V_4) * V_5) * V_6) * V_7
Reduction

Pthreads (low-level threading) has no reduction construct.

OpenMP's reduction construct allows only scalar types and simple operations.
Pipeline
Pipeline

Thread 0 sends alternately to Thread 1 and Thread 2.

Thread 3 receives alternately from Thread 1 and Thread 2.
Task Queue
Idioms

- Work sharing
- Reduction
- Pipeline
- Task queue
- Legacy
  - Obsolete: Making I/O or heap allocation thread safe

- Nondeterministic
  - Load balancing, random simulated interaction …
## SPLASH2

<table>
<thead>
<tr>
<th>Deterministic Constructs</th>
<th>barness</th>
<th>ffm</th>
<th>ocean</th>
<th>radiosity</th>
<th>raytrace</th>
<th>volrend</th>
<th>water-nsquared</th>
<th>water-spatial</th>
<th>cholesky</th>
<th>fft</th>
<th>lu</th>
<th>radix</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork/join</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>barrier</td>
<td>6</td>
<td>13</td>
<td>40</td>
<td>5</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>126</td>
</tr>
<tr>
<td>work sharing</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>reduction</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deterministic Idioms</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>work sharing</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>23</td>
<td>8%</td>
</tr>
<tr>
<td>reduction</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>21</td>
</tr>
<tr>
<td>pipeline</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>task queue</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>3%</td>
</tr>
<tr>
<td>legacy</td>
<td>1</td>
<td>15</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>28</td>
<td>10%</td>
<td></td>
</tr>
</tbody>
</table>

| nondeterministic         | 2      | 8   | -23   | 2         | 6        | -       | 2              | -             | -        | -   | -  | 43   | 16%   |

### Deterministic Constructs
- fork/join
- barrier
- work sharing
- reduction

### Deterministic Idioms
- work sharing
- reduction
- pipeline
- task queue
- legacy

### Total
- Deterministic Constructs: 20 tasks, 7%
- Deterministic Idioms: 23 tasks, 8%
- Nondeterministic: 43 tasks, 16%
# NPB-OMP

<table>
<thead>
<tr>
<th>Deterministic Constructs</th>
<th>BT</th>
<th>CG</th>
<th>DC</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
<th>UA</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork/join</td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>60</td>
<td>134</td>
</tr>
<tr>
<td>barrier</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>work sharing</td>
<td>37</td>
<td>20</td>
<td>-</td>
<td>1</td>
<td>8</td>
<td>11</td>
<td>71</td>
<td>16</td>
<td>38</td>
<td>78</td>
<td>280</td>
</tr>
<tr>
<td>reduction</td>
<td>-</td>
<td>6</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>4</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deterministic Idioms</th>
<th>BT</th>
<th>CG</th>
<th>DC</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
<th>UA</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>work sharing</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>reduction</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>80</td>
<td>89</td>
</tr>
<tr>
<td>pipeline</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>task queue</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>legacy</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>nondeterministic</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

538
# PARSEC

<table>
<thead>
<tr>
<th>Deterministic Constructs</th>
<th>blackscholes</th>
<th>bodytrack</th>
<th>facesim</th>
<th>ferret</th>
<th>fluidanimate</th>
<th>freqmine</th>
<th>raytrace</th>
<th>swaptions</th>
<th>vips</th>
<th>x264</th>
<th>canneal</th>
<th>dedup</th>
<th>streamcluster</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork/join</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>13</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>barrier</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>14</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>34</td>
<td>52</td>
</tr>
<tr>
<td>work sharing</td>
<td>2</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>28</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>reduction</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Deterministic Idioms     |              |           |         |        |              |          |          |           |      |       |         |       |               |       |              |
| work sharing             | -            | -         | -       | -      | 2            | -        | -        | -         | -    | 1     | -       | -     | 3             | 1%    |              |
| reduction                | -            | -         | -       | -      | 7            | -        | -        | -         | -    | -     | -       | -     | 7             | 3%    |              |
| pipeline                 | -            | -         | -       | -      | -            | -        | -        | -         | -    | -     | -       | 17    | 4             | 21    | 10%          |
| task queue               | -            | -         | 14      | 9      | -            | -        | 2        | -         | -    | -     | -       | 25    | 12%           |       |              |
| legacy                   | -            | -         | -       | -      | -            | -        | -        | -         | -    | -     | -       | -     | 0             | 0%    |              |
| nondeterministic         | -            | -         | -       | -      | 15           | -        | 6        | -         | -    | -     | -       | 21    | 10%           |       |              |
Aggregate

- Fork/Join: 17.87%
- Barrier: 14.79%
- Nondeterministic: 8.40%
- Legacy: 2.98%
- Task Queue Idioms: 3.62%
- Pipeline Idioms: 3.30%
- Reduction Idioms: 11.70%
- Work Sharing Idioms: 2.77%
- Reduction Constructs: 1.81%
- Work Sharing Constructs: 32.77%
OpenMP Benchmarks

- Fork/Join: 25.21%
- Barrier: 2.21%
- Pipeline Idioms: 0.85%
- Reduction Idioms: 16.35%
- Simple Reductions: 2.90%
- Work Sharing: 52.47%
Nondeterministic Synchronization

- Work Sharing Idioms: 8.44%
- Reduction Idioms: 35.71%
- Pipeline Idioms: 10.06%
- Task Queue Idioms: 11.04%
- Legacy: 9.09%
- Nondeterministic: 25.65%
Conclusions

• Deterministic parallel programming model compatible with many programs
• Reductions can help increase the number
Outline

- The Big Picture ✓
- Background ✓
- Analysis ✓
- Design and Semantics
- Implementation
- Evaluation
- Conclusion
Outline

• The Big Picture ✓
• Background ✓
• Analysis ✓
• Design and Semantics
• Extended Reduction
• Implementation
• Evaluation
• Conclusion
Foundations

- Workspace consistency
  - Memory consistency model
  - Naturally deterministic synchronization
- Working Copies Determinism
  - Programming model
  - Based on workspace consistency
“Parallel Swap” Example

x := 42
y := 33
(x,y) := (y,x)

Thread 0
x := 42
y := 33
x := y
x = y = 33

Thread 1
y := 33
y := x
x := y = 42
Memory Consistency Model

Communication Events

• Acquire
  • Acquires access to a location in shared memory
  • Involves a read

• Release
  • Enables access to a location in shared memory for other threads
  • Involves a write
Workspace Consistency

- Pair each release with a determinate acquire
- Delay visibility of updates until the next synchronization event
WC “Parallel Swap”
WC Fork/Join

Thread 0

(0,0) rel(1,0) → (1,0) acq(0,0)
(0,1) rel(2,0)
(0,2) rel(3,0)

Thread 1

(1,0) start

Thread 2

(2,0) acq(0,1)

Thread 3

(3,0) acq(0,2)

FORK

compute
compute
compute
compute

Thread 0

(0,3) acq(1,1) ← (1,1) rel(0,3)
(0,4) acq(2,1)
(0,5) acq(3,1)

JOIN

(2,1) rel(0,4)
(3,1) rel(0,5)

exit
exit
exit

Thread 1

(1,1) start

exit

Thread 2

(2,1) exit

exit

Thread 3

(3,1) exit

exit
WC Barrier

Thread 0

Thread 1

Thread 2

Thread 3

BARRIER

JOIN

FORK

Thread 0

Thread 1

Thread 2

Thread 3

(0,3) acq(1,1) (1,1) rel(0,3)

(0,4) acq(2,1) (2,1) rel(0,4)

(0,5) acq(3,1) (3,1) rel(0,5)

(0,0) rel(1,0) (1,0) acq(0,0)

(0,1) rel(2,0) (2,0) acq(0,1)

(0,2) rel(3,0) (3,0) acq(0,2)
while (true) {
    send(new_task(), out_1);
    send(next_task(), out_2);
    result = wait(in_1);
    store(result);
    result = wait(in_2);
    store(result);
}

while (true) {
    task = receive(in);
    result = process(task);
    send(result, out);
}
Nondeterministic Network
(For Contrast)

while(true) {
    result = receive(in);
    store(result);
    send(new_task(), out);
}

while(true) {
    task = receive(in);
    result = process(task);
    send(result, out);
}

Mute

X

Locks

Common channels
Working Copies Determinism

Shared memory

Fork: copy state

Thread A
A's writes

Thread B
B's writes
B reads “old” values

Join: merge changes
Conflicting writes $\rightarrow$ ERROR!
parent thread \rightarrow \text{working copy}
FORK

parent thread

working copy
parent thread

FORK

working copy

working copy

working copy

working copy

hide copy

reference copy

copy

copy

copy
parent thread

FORK

master

working copy

thread 1

working copy

thread n-1

working copy

JOIN

reference copy

hide copy

working copy

working copy

working copy
DOMP API

- Supports most OpenMP constructs
  - Parallel blocks
  - Work sharing
  - Simple (scalar-type) reductions
- Excludes OpenMP's few nondeterministic constructs
  - *atomic*, *critical*, *flush*
- Extends OpenMP with a generalized reduction
Example

// Multiply an n x m matrix A by an m x p matrix B
// to get an n x p matrix C.
void matrixMultiply(int n, int m, int p,
                    double ** A, double ** B, double ** C) {

    for (int i = 0; i < n; i++)
        for (int j = 0; j < p; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < m; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
// Multiply an n x m matrix A by an m x p matrix B
// to get an n x p matrix C.
void matrixMultiply(int n, int m, int p,
    double ** A, double ** B, double ** C) {

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < m; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
Example

// Multiply an n x m matrix A by an m x p matrix B
// to get an n x p matrix C.
void matrixMultiply(int n, int m, int p,
   double ** A, double ** B, double ** C) {
   #pragma omp parallel for
   for (int i = 0; i < n; i++)
      for (int j = 0; j < p; j++) {
         C[i][j] = 0.0;
         for (int k = 0; k < m; k++)
            C[i][j] += A[i][k] * B[k][j];
      }
}
Extended Reduction

- OpenMP's reduction is limited
  - Scalar types (no pointers!)
  - Arithmetic, logical, or bitwise operations
- Benchmark programmers used nondeterministic synchronization to compensate
Typical Workaround

In NPB-OMP EP (vector sum):

```
do 155 i = 0, nq - 1
  !$omp atomic
  q(i) = q(i) + qq(i)
155  continue
```
Typical Workaround

In NPB-OMP EP (vector sum):

```fortran
    do 155 i = 0, nq - 1
    !$omp atomic
    q(i) = q(i) + qq(i)
    155 continue
```

Nondeterministic programming model

Unpredictable evaluation order
DOMP Reduction API

- Binary operation \( op \)
  - Arbitrary, user-defined
  - Associative but not necessarily commutative
- Identity object \( idty \)
  - Defined in contiguous memory
- Reduction variable object \( var \)
  - Also defined in contiguous memory
- Size in bytes of \( idty \) and \( var \)
DOMP Reduction API

- Binary operation $op$
  - Associative but not necessarily commutative
- Identity object $idty$
  - Defined in contiguous memory
- Reduction variable object $var$
  - Also defined in contiguous memory
- Size in bytes of $idty$ and $var$

```c
void domp_xreduction(void(*op)(void*,void*), void** var, void* idty, size_t size);
```
Why the Identity Object?

- DOMP preserves OpenMP's guaranteed sequential-parallel equivalence semantics
- Each thread runs $op$ on the rhs and $idty$
- At merge time, each merging thread ("up-buddy") runs $op$ on its own and the other thread's (the "down-buddy's") version if $var$
- The master thread runs $op$ on the original $var$ and the cumulative $var$ from merges.
DOMP Replacement

In NPB-OMP EP (vector sum):

```
  do 155 i = 0, nq - 1
  !$omp atomic
     q(i) = q(i) + qq(i)
  155   continue
```

call xreduction_add(q_ptr, nq)

-------------------------------------------
void xreduction_add_(void ** input, int * nq_val) {
  nq = *nq_val;
  init_idty();
  domp_xreduction(&add_, input, (void *)idty,
                  nq * sizeof(double));
}
```
Desirable Future Extensions

- Pipeline
- Task Queue or Task Object
Desirable Future Extensions

- Pipeline
- Task Queue or Task Object

```c
#pragma omp sections pipeline
{ while (more_work()) {
    #pragma omp section
    { do_step_a(); }
    #pragma omp section
    { do_step_b(); }
    /* ... */
    #pragma omp section
    { do_step_n(); }
} }
```
Outline

- The Big Picture ✓
- Background ✓
- Analysis ✓
- Design and Semantics ✓
- Implementation
- Evaluation
- Conclusion
Outline

● The Big Picture √
● Background √
● Analysis √
● Design and Semantics √
● Implementation
● Evaluation
● Conclusion
Stats

- 8 files in libgomp
- ~ 5600 LOC
- Changes in gcc/omp-low.c and *.def files
  - To support deterministic simple reductions
Naive Merge Loop

for each data segment $seg$ in (stack, heap, bss)
    for each byte $b$ in $seg$
        $writer = \text{WRITER\_NONE}$
        for each thread $t$
            if ($seg[t][b] \neq \text{reference\_copy}[b]$)
                if ($writer \neq \text{WRITER\_NONE}$)
                    race condition exception()
                writer = $t$
        $seg[\text{MASTER}][b] = seg[\text{writer}][b]$
Improvements

- Copy on write (page granularity)
- Merge or copy pages only as needed
- Parallel merge (binary tree)
- Thread pool
Binary Tree Merge
Binary Tree Merge
Limitations

- Problem of granularity
  - False positive/false negative tradeoff
- Scaling constraints and space inefficiency
  - Global bookkeeping data structures
  - Globally visible heaps (mapped files)
- No nested parallelism
Outline

- The Big Picture ✓
- Background ✓
- Analysis ✓
- Design and Semantics ✓
- Implementation ✓
- Evaluation
- Conclusion
Outline

- The Big Picture ✓
- Background ✓
- Analysis ✓
- Design and Semantics ✓
- Implementation ✓
- Evaluation
- Conclusion
Speedup

![Graph showing speedup vs. threads for various benchmarks](image)

- Ideal
- DC
- EP
- swaptions
- BT
- MatMult
- Mandelbrot
- LU-cont
- LU-non-cont
- blackscholes
- FFT
- IS

20 September 2012

Amitai Aviram | Yale University CS
Why Is IS So Bad?

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Max Pages</th>
<th>Total Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMult</td>
<td>24578</td>
<td>24578</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BT</td>
<td>4</td>
<td>1911</td>
</tr>
<tr>
<td>DC</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>EP</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>IS</td>
<td>34778</td>
<td>90100</td>
</tr>
<tr>
<td>blackscholes</td>
<td>9768</td>
<td>9768</td>
</tr>
<tr>
<td>swaptions</td>
<td>677</td>
<td>677</td>
</tr>
<tr>
<td>FFT</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>LU-cont</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>LU-non-cont</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
# Converting Nondeterministic Code

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>DOMP Changes</th>
<th>Module</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMult</td>
<td>109</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>105</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BT</td>
<td>3589</td>
<td>16</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>DC</td>
<td>2809</td>
<td>3</td>
<td>48</td>
<td>2</td>
</tr>
<tr>
<td>EP</td>
<td>228</td>
<td>16</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>IS</td>
<td>634</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>blackscholes</td>
<td>359</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swaptions</td>
<td>1780</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FFT</td>
<td>1504</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LU-cont</td>
<td>2484</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LU-non-cont</td>
<td>1890</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Outline

- The Big Picture ✓
- Background ✓
- Analysis ✓
- Design and Semantics ✓
- Implementation ✓
- Evaluation ✓
- Conclusion
Outline

- The Big Picture √
- Background √
- Analysis √
- Design and Semantics √
- Implementation √
- Evaluation √
- Conclusion
Future Work

- More flexible design for changing the size of the thread pool at runtime
- Pipeline construct
- Task queue construct
- Nested parallelism
In Conclusion …

● Our analysis of benchmarks suggests that an accessible support framework for a deterministic parallel programming model may have wide applicability.

● Our experiments with DOMP suggest that such accessible deterministic parallel programming can be efficient and easy to use for many programs.
Thank You

- Bryan Ford
- Ramakrishna Gummadi
- Zhong Shao
- Emery Berger
- DeDiS Lab members
- Family and friends
- NSF Grant No. CNS-1017206.