DOMP: Deterministic OpenMP

Amittai Aviram and Bryan Ford
Decentralized/Distributed Systems Lab
Department of Computer Science
Yale University

Vancouver Systems Colloquium
University of British Columbia
7 October 2010
Deterministic Concurrency

Parallel program + same input → same output and behavior

- Reproduce any bug
- Replay computation exactly → Byzantine Fault Tolerance, peer-review accountability
- Address timing channel attacks? (CCSW '10)
The Underlying Problem

- Conventional programming models *inherently nondeterministic*
- Rely on *naturally nondeterministic* synchronization primitives: mutex locks, condition variables, ...
- Burden on the programmer to get synchronization right
- Low-level synchronization still allows high-level data races
High-Level Data Race

A
lock(x)
x := x * 2
unlock(x)

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

A gets lock first
x := 3x + 2

B gets lock first
x := 3x + 6

Heisenbug
Definition

*Naturally deterministic* synchronization:

Programming logic alone determines

- Which threads synchronize
- Where in each one's respective execution sequence they do so

Consequence:

*Timing* of arrival at synchronization points does not affect program behavior or output
Synchronization Abstractions

- Naturally deterministic
  - Fork/Join
  - Barrier
  - Future

- Naturally nondeterministic
  - Mutex lock
  - Condition variable
  - Semaphore
Our Goal

- *Naturally deterministic* programming model
- Programming abstractions expressive enough for most algorithms
- Runtime support that guarantees *race-free* deterministic execution
Previous Approaches

• **New languages**
  Dataflow languages, SHIM, Jade, DPJ, ...
  - Have to rewrite code
  - Some burden programmer with low-level permissions

• **Deterministic scheduling**
  DMP, CoreDet, Grace, Kendo, ...
  - Keeps underlying nondeterministic programming model
  - Data races reproducible but still there
Data Race Example

int x = 5;

// Start parallel execution here.
{
   // Thread A
   {
      if (input_is_as_usual)
      do_a_lot();
      x++;
   }
   // Thread B
   {
      x++;
   }
}

• Programmer forgets to synchronize
• Tests run great!
• On “unusual” input, deterministic scheduler may always give 6 😞
• We want this code never to work!
Working-Copies Determinism

- Data like documents in version control system
- Fork-join parallelism model
- At fork: runtime gives each concurrent thread a working copy of state (like “checkout”)
- Concurrent threads are isolated
- At barrier and join: runtime merges copies
- Two writes to same location → ERROR!
Thread A

A's writes

Thread B

B's writes

B reads “old” values

Shared memory

Fork: copy state

Join: merge changes

Conflicting writes → ERROR!
Determinator

• OS kernel based on working-copies determinism (OSDI '10)
• Race-free processes, threads, I/O
• Threading API limited to pthread-like fork/join and barrier
• Need for more expressive API
Next Step: Better API

Starting point: adapt OpenMP

- Expressive parallel programming API
- Already well-established
- Mostly compatible with Determinator model
This Talk

- Background ✓
- OpenMP and DOMP Basics
- Is DOMP Practical?
- Challenges and possibilities
- Conclusion
• Background

• **OpenMP and DOMP Basics**

• Is DOMP Practical?

• Challenges and possibilities

• Conclusion
OpenMP

- **Annotations** on a sequential program
- Legacy languages—little (no) rewriting
- Directives annotate structured blocks
  - `parallel`—general fork/join
  - `for`—parallel loop execution
  - `sections`—parallel tasks
- Optional clauses modify default behavior
  - `shared, private`, etc. for variables
  - `reduction` (sum, product, ...) across threads
Sequential Version

// 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 ** restrict A, double ** restrict B,
        double ** restrict 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];
        }
}
OpenMP Version

// 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 ** restrict A, double ** restrict B,
                double ** restrict 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];
        }
}
OpenMP Semantics

```c
void matrixMultiply(int n, int m, int p,
                     double ** restrict A, double ** restrict B,
                     double ** restrict 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];
    }
}
```

- Starts with a single parent thread
- Creates new threads, distributes work
- Joins threads to parent
Reductions

- Sum, product, ... on the *same variable* across threads
- Lock-free safety from data races
- Results available only after relevant parallel block
Reduction Clause in Sections

```c
int x = 5;
#pragma omp parallel sections reduction(+: x)
{
    #pragma omp section
    {
        if (input_is_as_usual)
            do_a_lot();
        x++;
    }
    #pragma omp section
    {
        x++;
    }
    #pragma omp section
    {
        x++;
    }
}
printf("x = %d\n", x) // Always prints 7!
```
Reduction Semantics

```c
int x = 5;
#pragma omp parallel sections reduction(+: x)
{
    #pragma omp section
    {
        if (input_is_as_usual)
            do_a_lot();
        x++;
    }
    #pragma omp section
    {
        do_a_little();
        x++;
    }
    #pragma omp section
    {
        do_a_little();
        x++;
    }
}
printf("x = %d\n", x) // Always prints 7 !
```

*sections* assigns each *section* to a different thread

*reduction* aggregates the + on *x* across sections/threads
DOMP = Deterministic OpenMP

- Includes compatible features
  - Fork-join parallelism, structured blocks
  - Core directives—naturally deterministic
  - Reductions—naturally deterministic
- Excludes incompatible features
  - atomic, critical, mutex, etc.
  - Many programs don't need these
  - Reductions may help avoid them
DOMP Semantics

// 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 ** restrict A, double ** restrict B,
    double ** restrict 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];
    }
}

Merges copies of shared vars into parent's vars (if no data race)

Distributes copies of shared state
• Background
• OpenMP and DOMP Basics
• **Is DOMP Practical?**
• Challenges and possibilities
• Conclusion
• Background
• OpenMP and DOMP Basics

**Is DOMP Practical?**

- From the programmer's POV
  Parallel programming without nondeterministic abstractions

- From the user's POV
  Acceptable performance and scalability

• Challenges and possibilities
• Conclusion
Parallel Programming Without Locks?

- Programmers often use low-level nondeterministic abstractions to build higher-level deterministic abstractions
- Examples: SPLASH, NPB
- Nondeterministic abstractions should be reserved for nondeterministic algorithms
Synchronization in SPLASH

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Deterministic Primitives (56%)</th>
<th>Nondeterministic Primitives (44%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fork/join</td>
<td>barrier</td>
</tr>
<tr>
<td>barnes</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>ffm</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>ocean</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>radiosity</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>volrend</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>water-nsquared</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>water-spatial</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>cholesky</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>fft</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>lu</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>radix</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Proportion</td>
<td>7.11%</td>
<td>49.37%</td>
</tr>
</tbody>
</table>

Nondeterministic algorithm
Synchronization in NPB-OMP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Deterministic Primitives (70%)</th>
<th>Nondeterministic Primitives (30%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>fork/join</td>
<td>barrier</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>flush (barrier)</td>
</tr>
<tr>
<td>CG</td>
<td>14</td>
<td>atomic (reduction)</td>
</tr>
<tr>
<td>DC</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>IS</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>LU</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>LU-HP</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>UA</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>Proportion</td>
<td>67.74%</td>
<td>2.30%</td>
</tr>
<tr>
<td></td>
<td>4.61%</td>
<td>24.88%</td>
</tr>
</tbody>
</table>

All nondeterministic abstractions used to build deterministic abstractions.
How Programs Use Synchronization

- fork/join: 36.04%
- barrier: 27.03%
- lock (work sharing): 13.63%
- flush (barrier): 2.20%
- lock/atomic (reduction): 16.48%
- lock (time step update): 1.54%
- lock (load balancing): 3.08%
- Nondeterministic algorithm: 1.08%

7 Oct 2010
Vancouver Systems Colloquium | UBC
Can it be Efficient?

- Problem:
  Cost of copying and merging data:
  \(O(num\_threads \times bytes\_of\_data)\)

- Solution:
  Lazy per-page copy-on-write
  Lazy page-granularity merge
Experience

- *Dthreads, Determinator*
- Good results for some benchmarks
- Great for “embarrassingly parallel” applications
- Fine-grained parallelism can be expensive
Determinator Results

The graph illustrates the speedup over Linux for various benchmarks under different numbers of CPUs. The x-axis represents the benchmarks, and the y-axis shows the speedup ratio. Different colors and bars indicate the performance with 1, 2, 4, 8, and 12 CPUs, respectively.
Challenges

• Reductions
  – Extend to user-defined operations?
  – Must they be associative and commutative?
• How much real code can fit the DOMP model?
Future Work

- Deterministic file I/O?
- Non-hierarchical parallelism?
  - Pipelining, work queues
  - Task/taskwait directive
- Optimizations exploiting type safety?
Conclusion

- DOMP: proposed race-free, deterministic programming framework
- API based on OpenMP
- Applies working-copies approach to enforce determinism
- Combines expressiveness, reliability, and efficiency
Thank you

- Shu-Chun Weng
- Sen Hu
- Y. Richard Yang
- Ramakrishna Gummadi
- James Aspnes