

# 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)

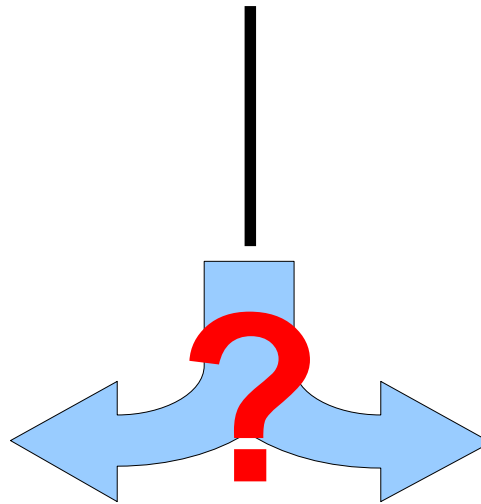
# The Underlying Problem

- 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

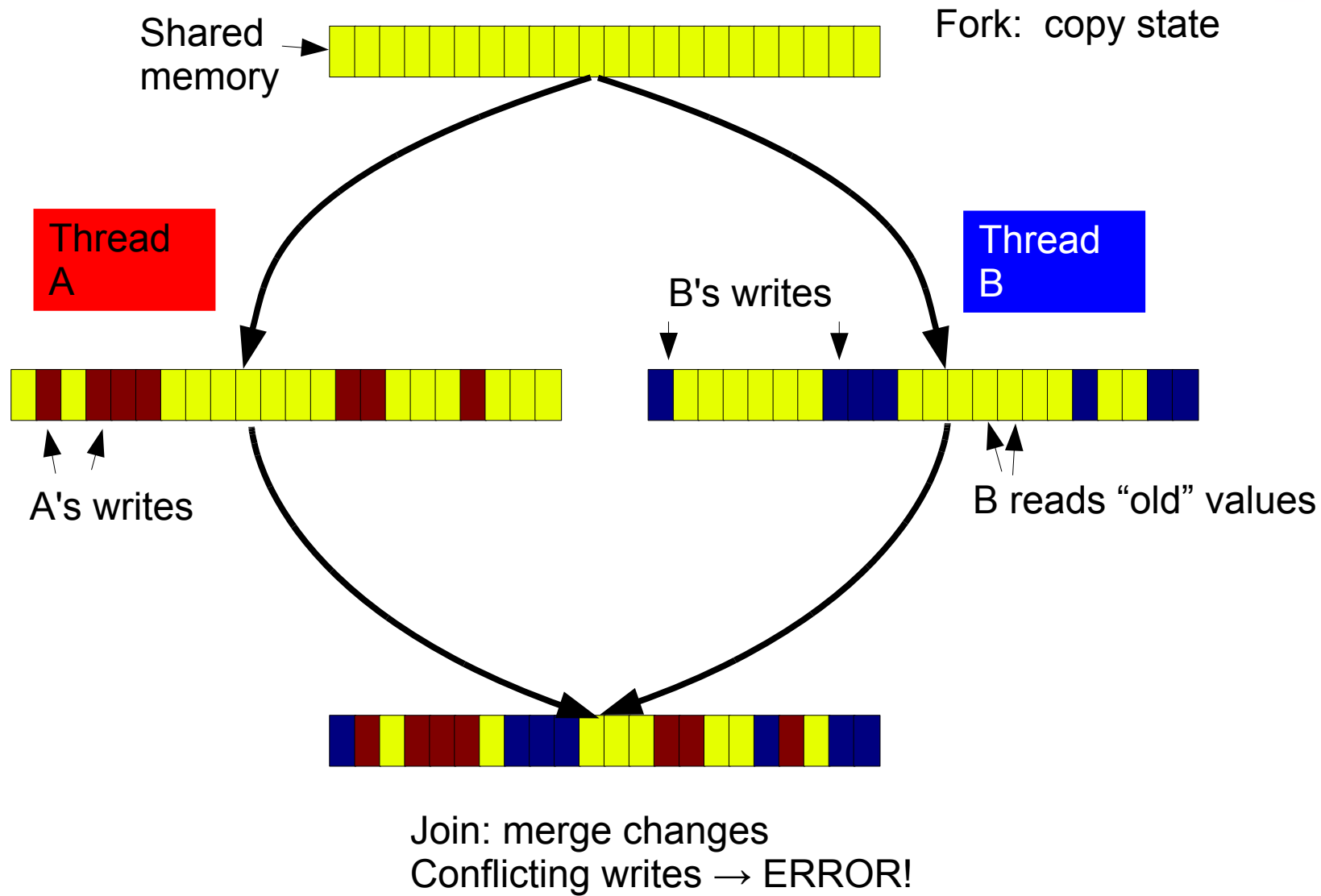
```
int x = 5;

// 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!



# 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

```
// 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];  
        }  
    }  
}
```

Starts with  
single parent  
thread

Joins threads  
to parent

Creates  
new threads,  
distributes work

# 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

```
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++;
    }
}
printf("x = %d\n", x) // Always prints 7!
```

# Reduction Semantics

```
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++;
    }
}
```

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

*reduction* aggregates the + on x across sections/threads

`printf("x = %d\n", x) // Always prints 7 !`

# 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];  
        }  
}
```

Distributes copies  
of shared state

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

- 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

Benchmark	Deterministic Primitives (56%)		Nondeterministic Primitives (44%)			
	fork/join	barrier	lock (work sharing)	lock (reduction)	lock (time step update)	lock (load balancing)
barnes	1	6	6			
fmm	1	13	28			
ocean	1	40	2	2		
radiosity	3	5	5	5		14
volrend	5	13	8	6		
water-nsquared	1	9	1	4	3	
water-spatial	1	9	2	4	4	
cholesky	1	4	7			
fft	1	7	1			
lu	1	5	1			
radix	1	7	1			
Proportion	7.11%	49.37%	25.94%	8.79%	2.93%	5.86%

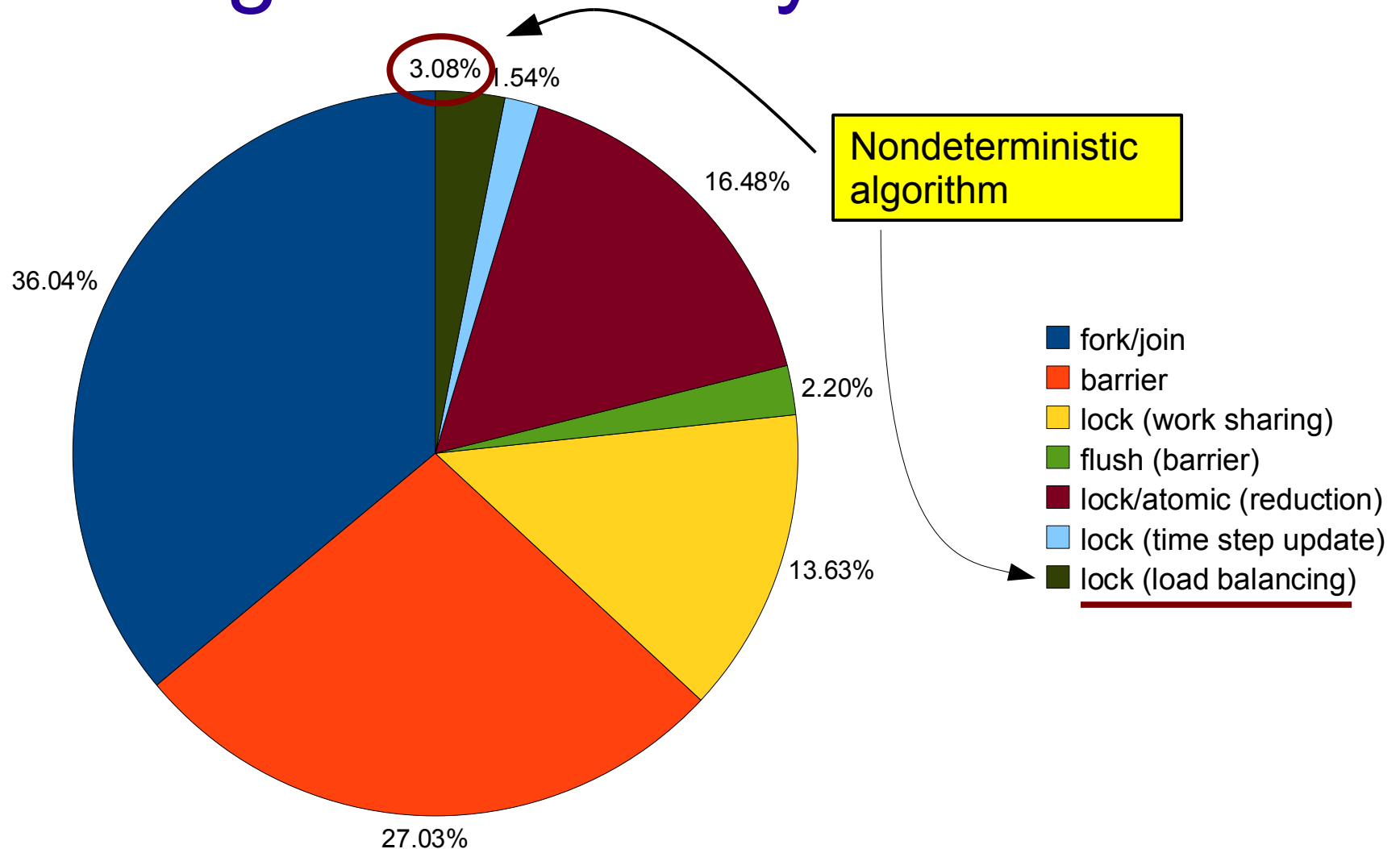
Nondeterministic algorithm

# Synchronization in NPB-OMP

Benchmark	Deterministic Primitives (70%)		Nondeterministic Primitives (30%)	
	fork/join	barrier	flush (barrier)	atomic (reduction)
BT	10			1
CG	14			
DC	2			
EP	2			1
FT	8			
IS	3	2		
LU	9	3	10	2
LU-HP	15			2
MG	11			
SP	14			2
UA	59			46
Proportion	67.74%	2.30%	4.61%	24.88%

*All nondeterministic abstractions used to build deterministic abstractions.*

# How Programs Use Synchronization



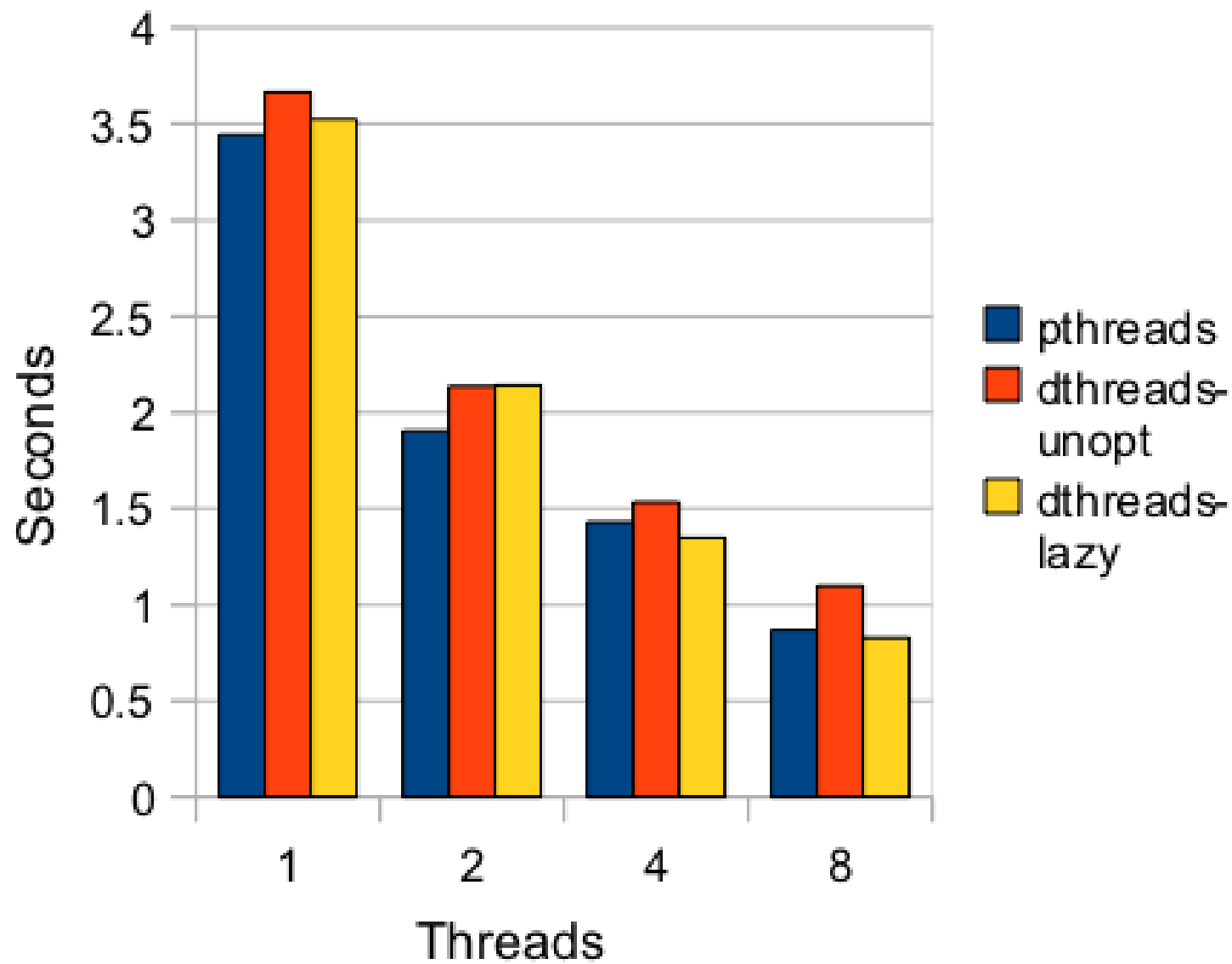
# Can it be Efficient?

- Problem:  
Cost of copying and merging data:  
 $O(\text{num\_threads} \times \text{bytes\_of\_data})$
- Solution:  
Lazy per-page copy-on-write  
Lazy page-granularity merge

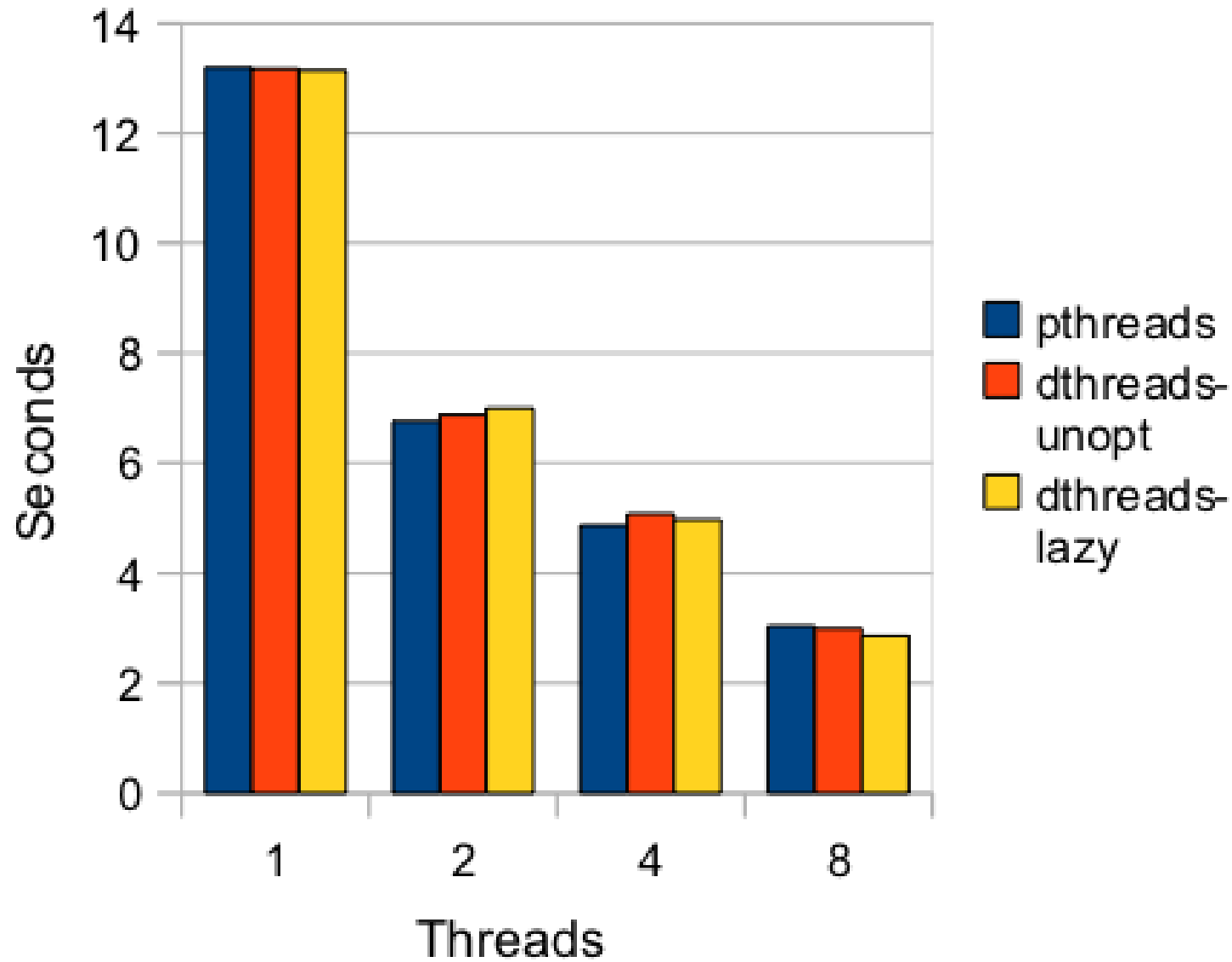
# Experience

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

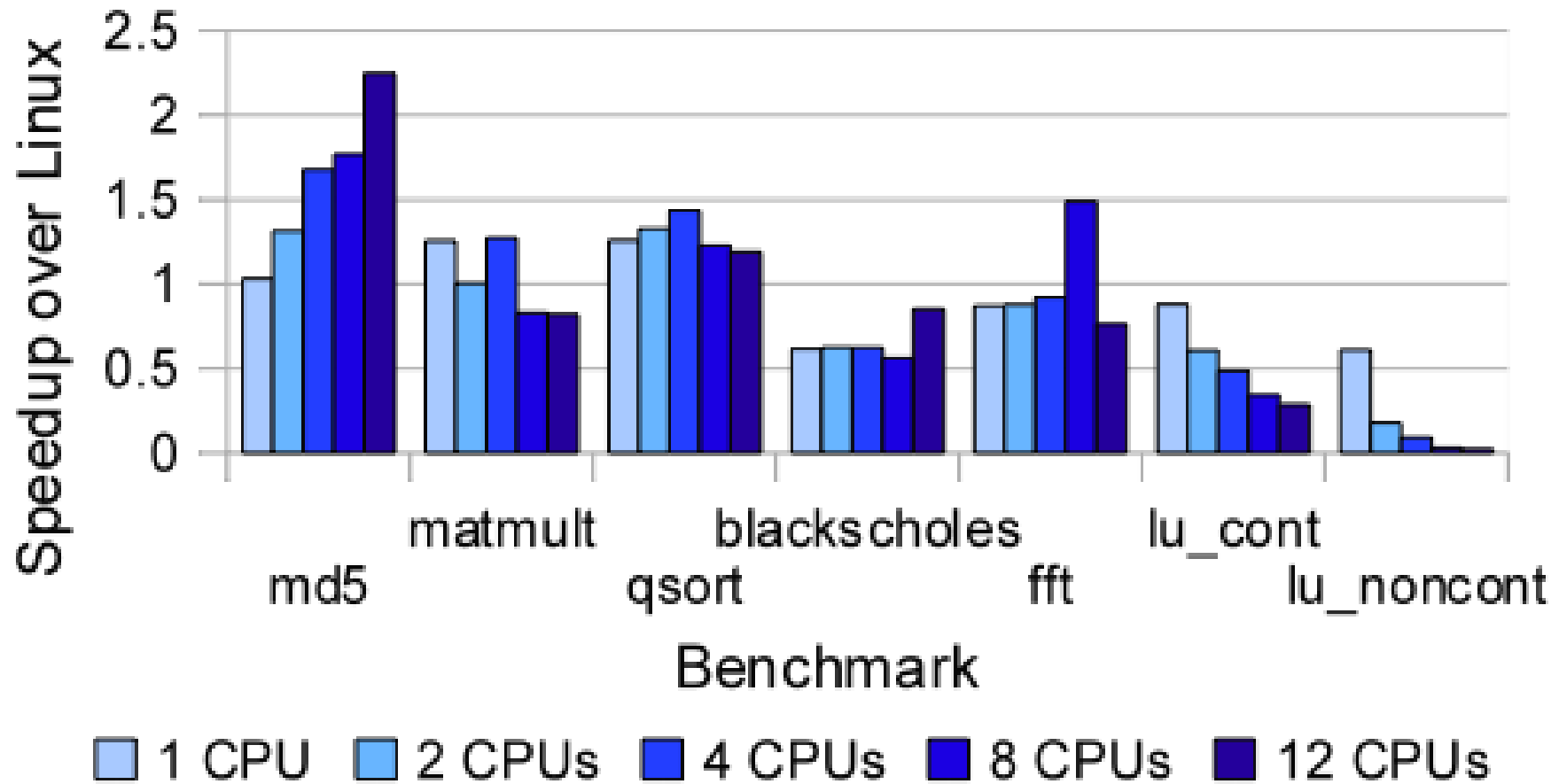
# Blackscholes



# Swaptions



# Determinator Results



# 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