Deterministic OpenMP For Race-Free Parallelism

Amittai Aviram and Bryan Ford
Decentralized/Distributed Systems (DeDiS)
Department of Computer Science
Yale University

HotPar 2011
Berkeley, CA
26 May 2011

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)
- Build an end-to-end verifiable system

The Underlying Problem

- Conventional programming models *inherently nondeterministic*
- Rely on *naturally nondeterministic* synchronization primitives: mutex locks, condition variables, ...

Definition (WoDet '11)

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

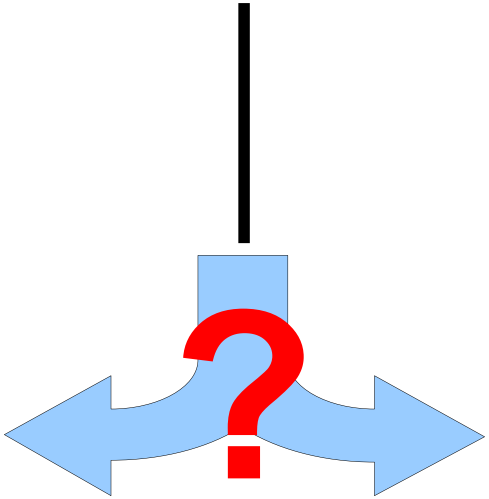
Naturally Nondeterministic

- Mutex lock
- Condition variable
- Semaphore

Naturally Nondeterministic Synchronization Causes Problems

- Burdens the programmer to get synchronization right
- Even when correct, 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 := 2x + 3

B gets lock first
x := 2x + 6

Synchronization Abstractions

Naturally Deterministic

- Fork/join
- Barrier
- Future

Safe

Naturally Nondeterministic

- Mutex lock
- Condition variable
- Semaphore

Risky

Our Goal

- *Naturally deterministic* programming model
- **Let the programmer live in a deterministic world**
- Deterministic programming abstractions expressive enough for most algorithms
- Runtime support that guarantees *race-free* deterministic execution

This Talk

- The Goal √
  A Naturally Deterministic Programming Model
- Background & Related Work
- From OpenMP to DOMP Semantics
- DOMP Runtime
- Efficiency
- Conclusion

Background & Related Work

Previous Approaches

- New languages
  Dataflow languages, SHIM, Jade, DPJ, ...
  - Have to rewrite code
- Deterministic scheduling
  DMP, CoreDet, Grace, Kendo, …
  - Keeps underlying nondeterministic programming model
  - Data races reproducible but not eliminated

Deterministic Scheduling & Races

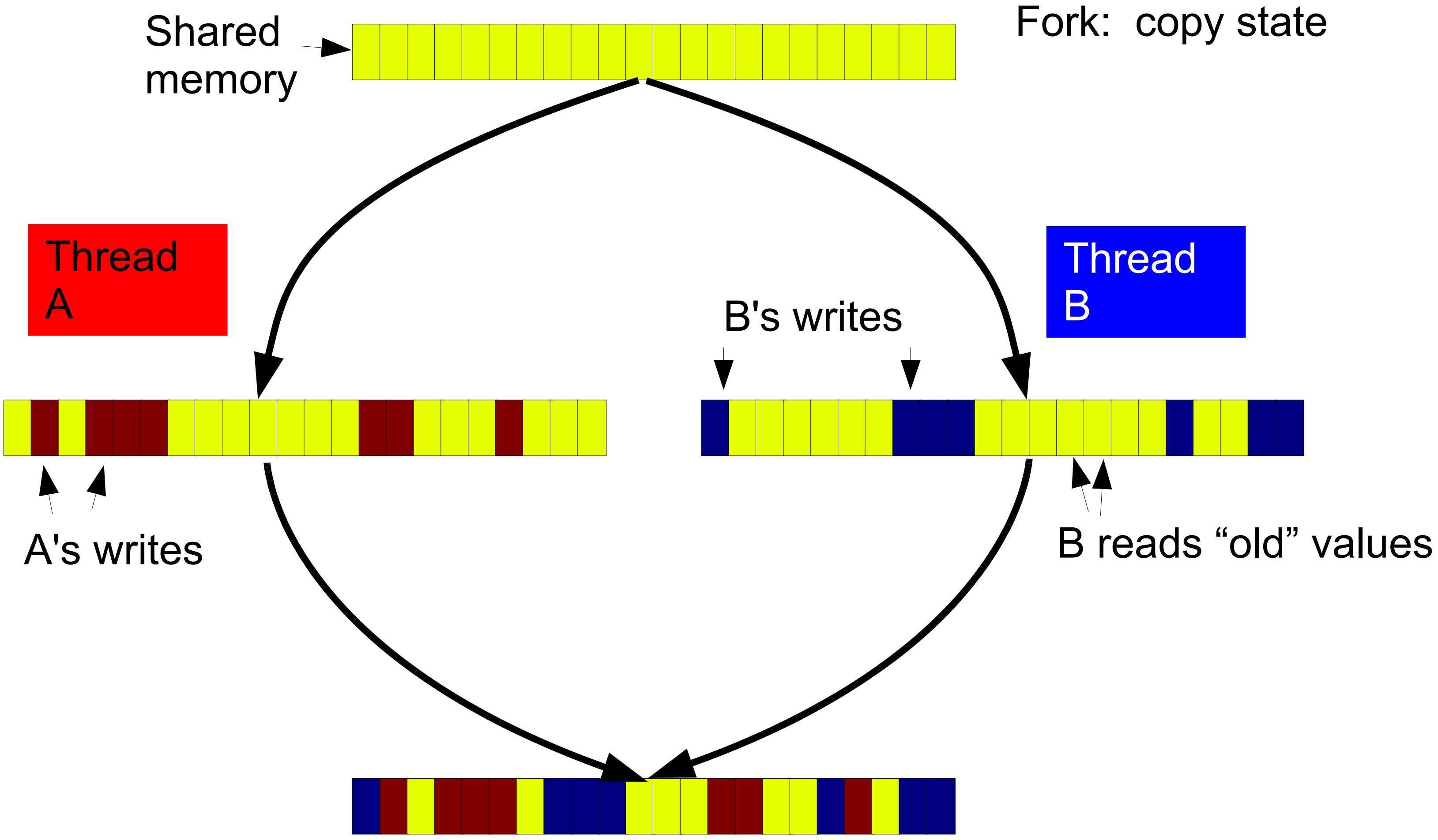
```
int x = 5;

// 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

(WoDet '11)

- Data like documents in version control system
- Fork-join parallelism model—naturally deterministic semantics
- 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!



Shared memory

Fork: copy state

Thread A

Thread B

B's writes

A's writes

B reads “old” values

Join: merge changes
Conflicting writes → ERROR!

Determinator

- OS kernel based on working-copies determinism (OSDI '10)
- Race-free processes, threads, I/O
- **Naturally deterministic threading API**, but
- Limited to pthread-like fork/join and barrier
- Need for more expressive API

A Better API

Starting point: OpenMP

Attractive because:

- Expressive parallel programming API
- Already well-established
- Most features already naturally deterministic
  - Mostly compatible with working-copies determinism

But, ...

Why OpenMP Is Not the Answer

OpenMP includes naturally nondeterministic synchronization abstractions

- *atomic*
- *critical*
- *flush*

Why OpenMP Is Not the Answer

OpenMP includes naturally nondeterministic synchronization abstractions

- *atomic*
- *critical*
- *flush*

We would like simply to disallow these, but

Why OpenMP Is Not the Answer

OpenMP includes naturally nondeterministic synchronization abstractions

- *atomic*
- *critical*
- *flush*

We would like simply to disallow these, but

Programmers find need to rely on them!

Key Insight

- As popular benchmark suites show, programmers rely on nondeterministic primitives—but
- In most cases, they use them to implement *deterministic* higher-level idioms
- We need *deterministic* high-level abstractions to express these idioms
- Need to *extend* OpenMP to fill these gaps

Primitives versus Idioms

- Primitives (in OpenMP, pthreads, etc.)
  - Deterministic: fork/join, barrier, OpenMP work-sharing
  - Nondeterministic: locks, condition variables, *atomic*, *critical*, *flush* …
- Idioms
  - Deterministic: reduction, pipeline ad-hoc work-sharing
  - Nondeterministic: load balancing, task queues, work stealing, user-level scheduling

Primitives versus Idioms

- Primitives (in OpenMP, pthreads, etc.)
  - Deterministic: fork/join, barrier, work-sharing (OpenMP only)
  - Nondeterministic: locks, condition variables, *atomic*, *critical*, *flush* …
- Idioms
  - **~92%** Deterministic: work-sharing, reduction, pipeline
  - **~8%** Nondeterministic: load balancing, task queues, work stealing, user-level scheduling

Code Analysis

- Manually inspect & analyze SPLASH, NPB-OMP, and PARSEC benchmarks
- Count and classify all uses of synchronization primitives
- Classify uses of *nondeterministic* primitives by *idiom* they are used to build

Synchronization in SPLASH

	Deterministic Primitives (56%)		Nondeterministic Primitives (44%)			
	fork/join	barrier	work sharing idioms	reduction idioms	pipeline idioms	load balancing
barnes	1	6	6			
fmm	1	13	28			
ocean	1	40	2	2		
radiosity	3	5	2	8		21
volrend	5	13	8	6		
water-nsquared	1	9	1	7		
water-spatial	1	9	1	4	2	
cholesky	1	4	1			2
fft	1	7	1			
lu	1	5	1			
radix	1	7	1			
	7.11%	49.37%	21.76%	11.30%	0.84%	9.62%

The **only** nondeterministic idioms

Synchronization in NPB-OMP

	Deterministic Primitives (70%)		Nondeterministic Primitives (30%)	
	fork/join	barrier	reduction idioms	pipeline idioms
BT	10		1	
CG	14			
DC	2			
EP	2		1	1
FT	8			
IS	3	2		
LU	9	3	2	10
LU-HP	15		2	
MG	11			
SP	14		2	
UA	59		44	
	68.69%	2.34%	24.30%	4.67%

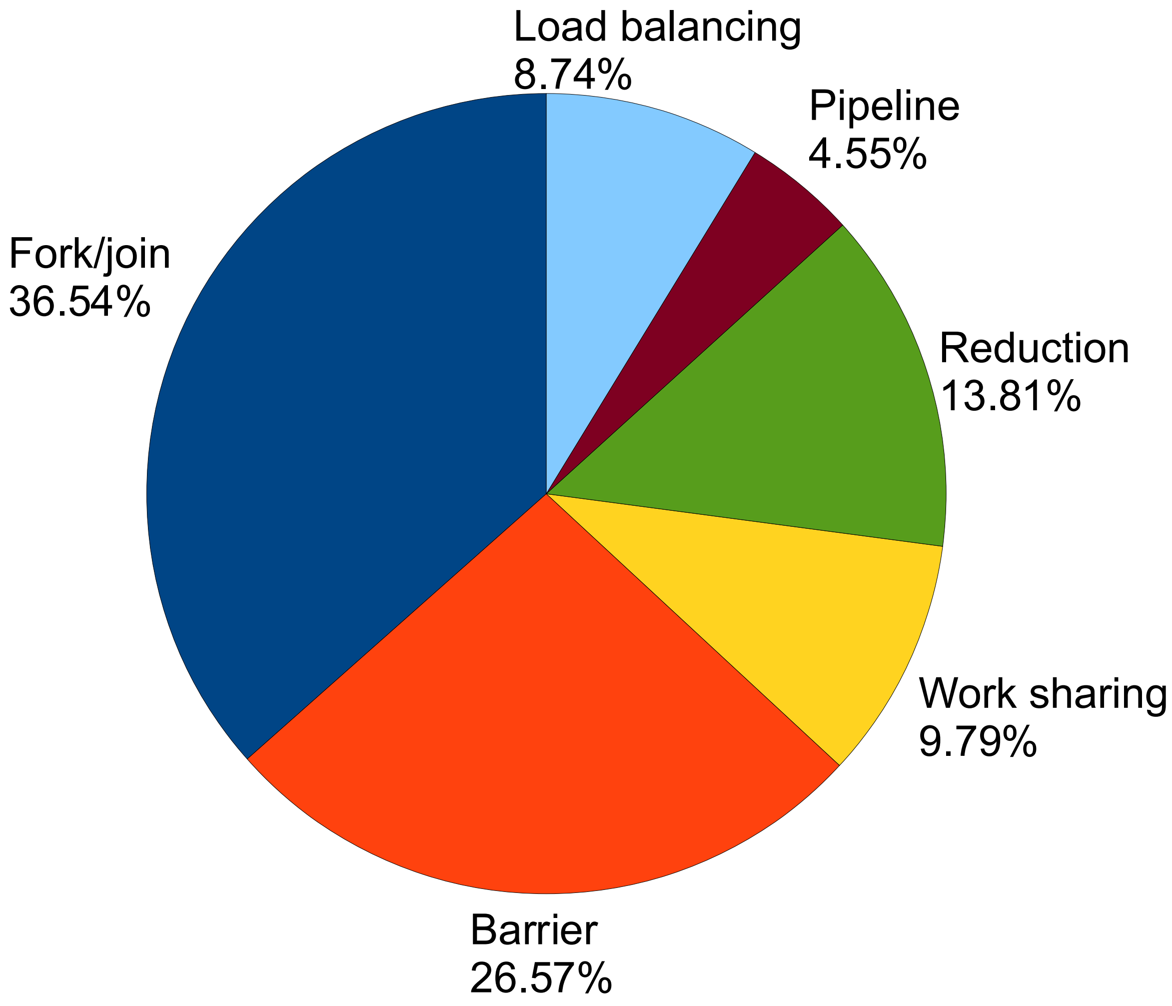
All nondeterministic abstractions used to build deterministic abstractions.

Synchronization in PARSEC

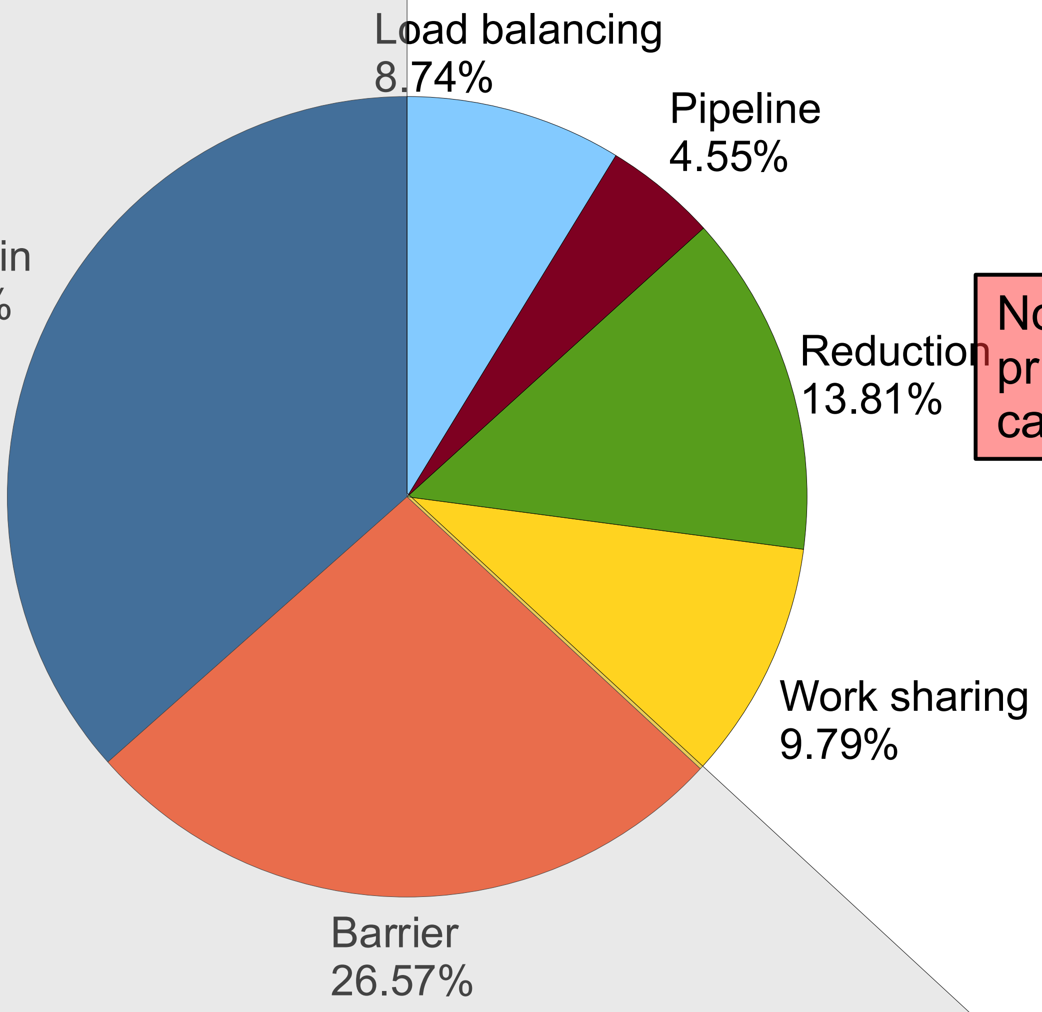
	Deterministic Primitives (62%)		Nondeterministic Primitives (38%)			
	fork/join	barrier	work sharing idioms	reduction idioms	pipeline idioms	load balancing
blackscholes	1					
bodytrack	4					
canneal	1	1	1			
dedup	1	1	1		10	
facesim	2					10
ferret	7					
fluidanimate	13					
freqmine	7					2
raytrace	1	1				
streamcluster	1	27	3			
swaptions						
vips						15
x264	2				4	
	37.82%	24.37%	3.36%	0.0%	11.76%	22.69%

The **only** nondeterministic idioms

How Programs Use Synchronization



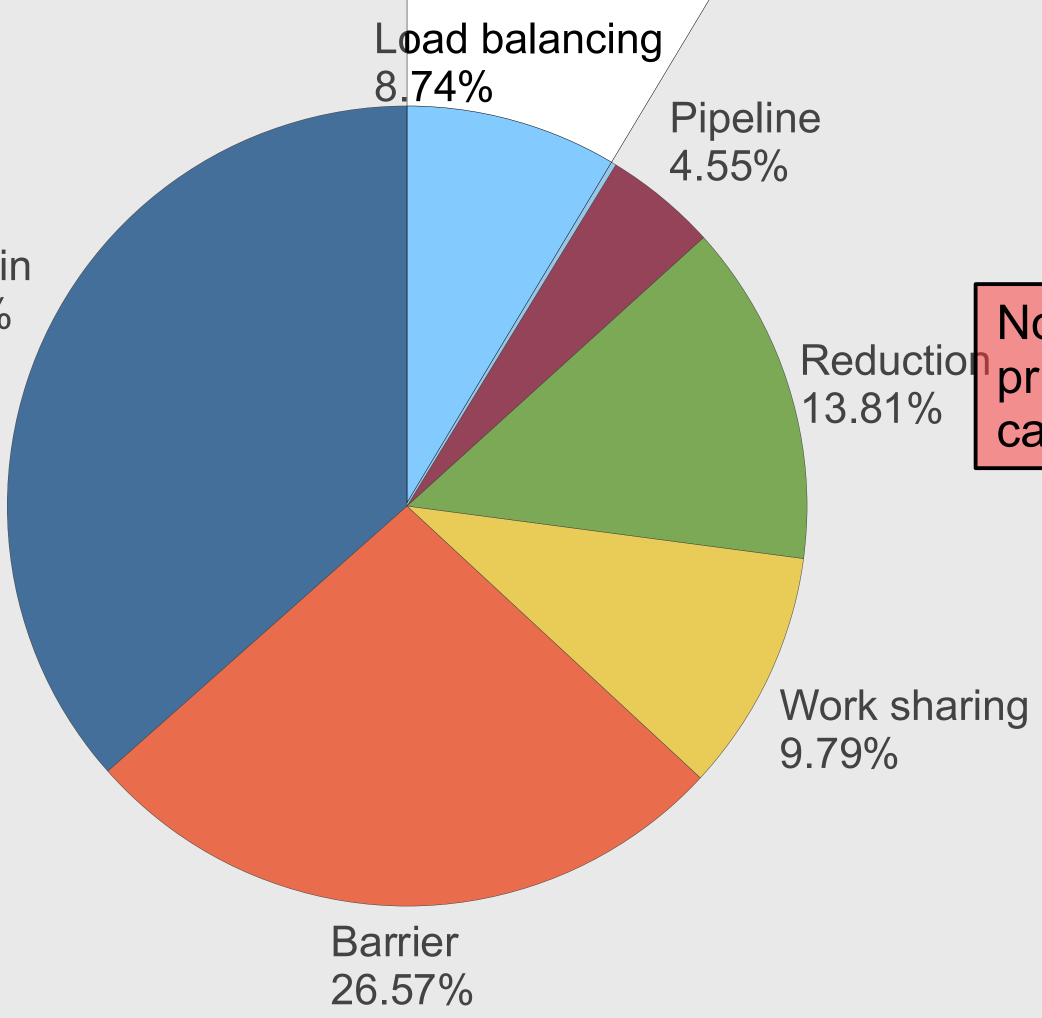
How Programs Use Synchronization



Deterministic primitives

Nondeterministic primitives categorized by use

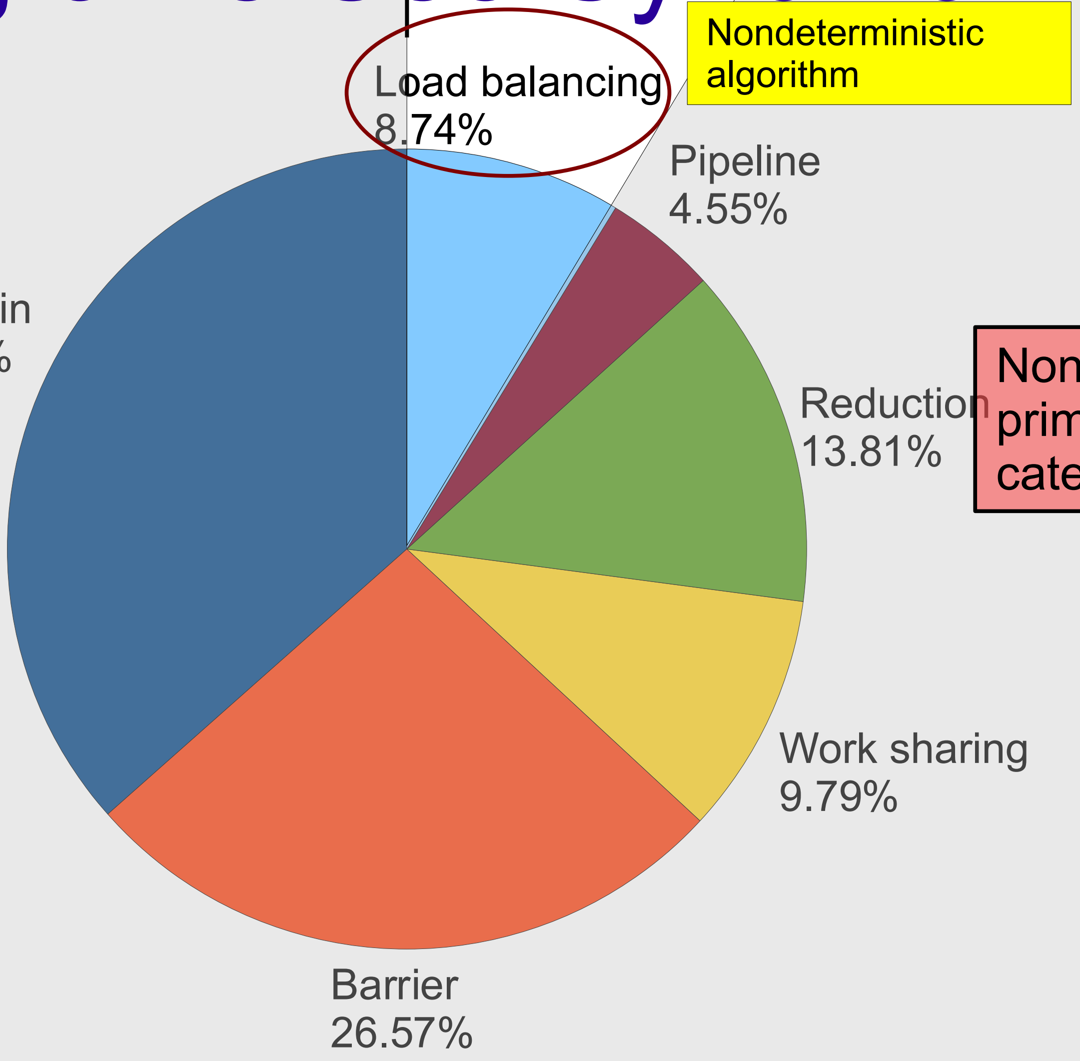
How Programs Use Synchronization



Deterministic primitives

Nondeterministic primitives categorized by use

How Programs Use Synchronization

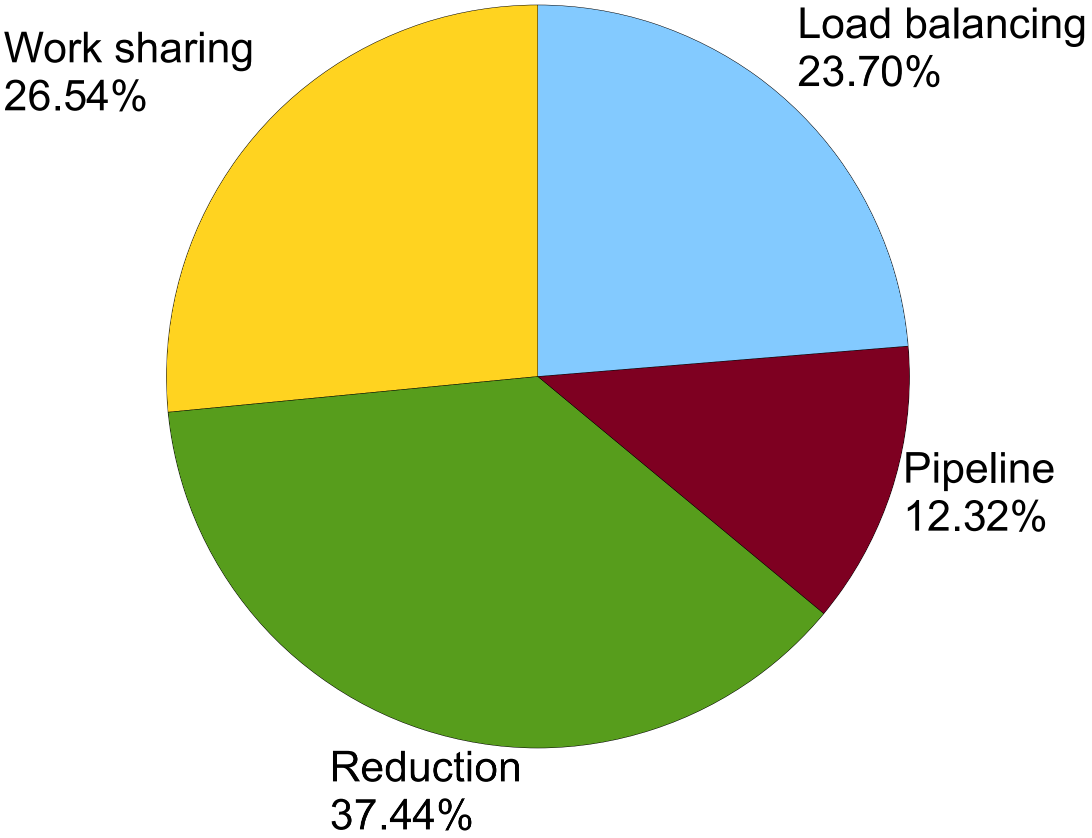


Nondeterministic algorithm

Deterministic primitives

Nondeterministic primitives categorized by use

Uses of Nondeterministic Primitives



Study Conclusion

Many (most) parallel programs could be expressed exclusively in a *naturally determistic* API if it includes abstractions for common high-level deterministic idioms.

OpenMP → DOMP!

- The Goal √
- Background & Related Work √
- From OpenMP to DOMP Semantics
- DOMP Runtime
- Efficiency
- Conclusion

From OpenMP to DOMP

Deterministic OpenMP (DOMP)

- Redefine compatible OpenMP constructs to be explicitly deterministic
- Offer deterministic alternatives to nondeterministic OpenMP constructs

OpenMP

- *Annotations* to parallelize 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 ** 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];
            }
}
```

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 ** 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];
            }
}
```

OpenMP Semantics

```
// Multiply an n x m matrix A by an m x p matrix B
// to get an n x p matrix C.
  id 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];
          }
}
```

Starts with
single parent
thread

OpenMP Semantics

```
// Multiply an n x m matrix A by an m x p matrix B
// to get an n x p matrix C.
  id 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];
            }
}
```

Starts with single parent thread

Creates new threads, distributes work

OpenMP Semantics

```
// Multiply an n x m matrix A by an m x p matrix B
// to get an n x p matrix C.
oid 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];
               }
}
```

Starts with
single parent
thread

Creates
new threads,
distributes work

Joins threads
to parent

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 ** 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];
            }
}
```

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 ** 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];
                  }
}
```

Distributes copies of shared state

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 ** 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];
            }
}
```

Distributes copies of shared state

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

OpenMP Reductions

- Sum, product, ... on the *same variable* across threads
- Lock-free safety from data races
- Results available only after relevant parallel block
- NATURALLY DETERMINISTIC!

OpenMP Reduction

```
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!
```

OpenMP Reduction

```
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!
```

sections assigns each *section* to a different thread

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

Unfortunately, ...

Not general enough

- Arithmetic or logical operations only
- Scalar types only
- Commutative and associative only

… Forcing the programmer to resort to hand-rolling reductions out of *naturally nondeterministic* primitives

DOMP General Reduction

```
// Assume: dimensions n, m, p have been set globally.
// Returns the product of two matrices.
extern matrix * matrix_multiply(matrix * A, matrix * B);

matrix * I = new_identity_matrix(n, m);
matrix * C = matrices[0];

#pragma omp parallel for reduction(matrix_multiply : I : C)
for (int i = 1; i < NUM_MATRICES; i++)
      C = matrix_multiply(C, matrices[i]);
// C now points to the product of all the matrices in the array matrices.
```

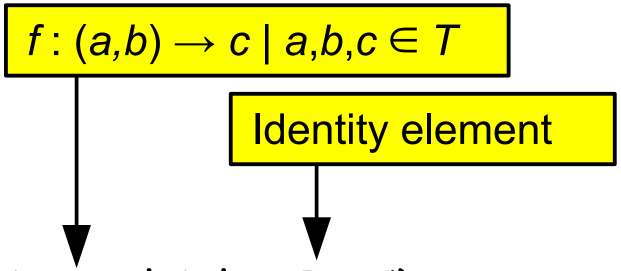
DOMP General Reduction

```
// Assume: dimensions n, m, p have been set globally.
// Returns the product of two matrices.
extern matrix * matrix_multiply(matrix * A, matrix * B);

matrix * I = new_identity_matrix(n, m);
matrix * C = matrices[0];

#pragma omp parallel for reduction(matrix_multiply : I : C)
for (int i = 1; i < NUM_MATRICES; i++)
    C = matrix_multiply(C, matrices[i]);
// C now points to the product of all the matrices in the array matrices.
```

$f : (a,b) \to c \mid a,b,c \in T$

DOMP General Reduction

```
// Assume: dimensions n, m, p have been set globally.
// Returns the product of two matrices.
extern matrix * matrix_multiply(matrix * A, matrix * B);

matrix * I = new_identity_matrix(n, m);
matrix * C = matrices[0];

#pragma omp parallel for reduction(matrix_multiply : I : C)
for (int i = 1; i < NUM_MATRICES; i++)
      C = matrix_multiply(C, matrices[i]);
// C now points to the product of all the matrices in the array matrices.
```

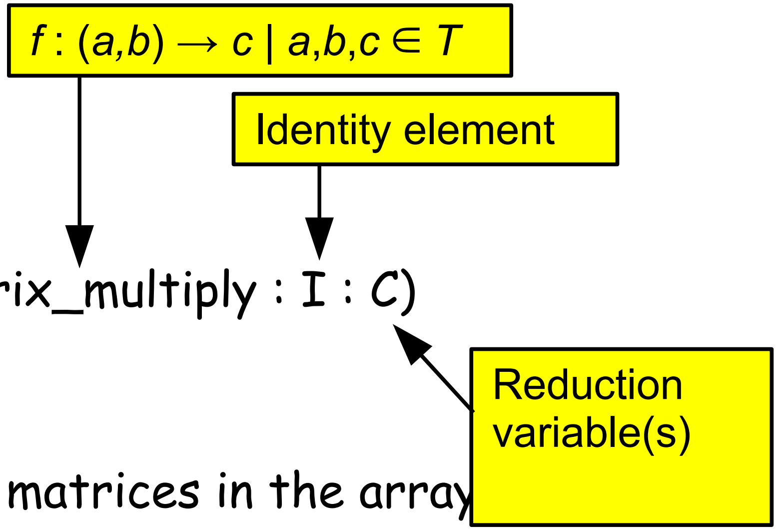


$f : (a,b) \rightarrow c \mid a,b,c \in T$

Identity element

DOMP General Reduction

```
// Assume: dimensions n, m, p have been set globally.
// Returns the product of two matrices.
extern matrix * matrix_multiply(matrix * A, matrix * B);

matrix * I = new_identity_matrix(n, m);
matrix * C = matrices[0];

#pragma omp parallel for reduction(matrix_multiply : I : C)
for (int i = 1; i < NUM_MATRICES; i++)
      C = matrix_multiply(C, matrices[i]);
// C now points to the product of all the matrices in the array
```



$f : (a,b) \rightarrow c \mid a,b,c \in T$

Identity element

Reduction variable(s)

Pipelines

Another deterministic idiom programmers hand-roll from naturally nondeterministic primitives for lack of high-level abstractions

E.g. LU (NPB-OMP): Uses ad hoc synchronization (*flush* memory barrier) to busy-wait on a flag

DOMP Pipelines

```
#pragma omp sections pipeline
while x = (more_work(x)) {
        #pragma omp section
        {
                x = do_step_a(x);
        }
        #pragma omp section
        {
                x = do_step_b(x);
        }
        /* … */
        #pragma omp section
        {
                x = do_step_n(x);
        }
}
```

DOMP Pipelines

```
#pragma omp sections pipeline
while x = (more_work(x)) {
        #pragma omp section
        {
                x = do_step_a(x);
        }
        #pragma omp section
        {
                x = do_step_b(x);
        }
        /* … */
        #pragma omp section
        {
                x = do_step_n(x);
        }
}
```

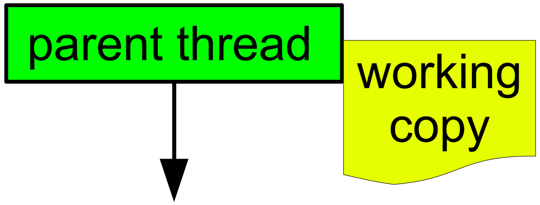
One thread per *section*

Each *section* starts executing when the one before has finished

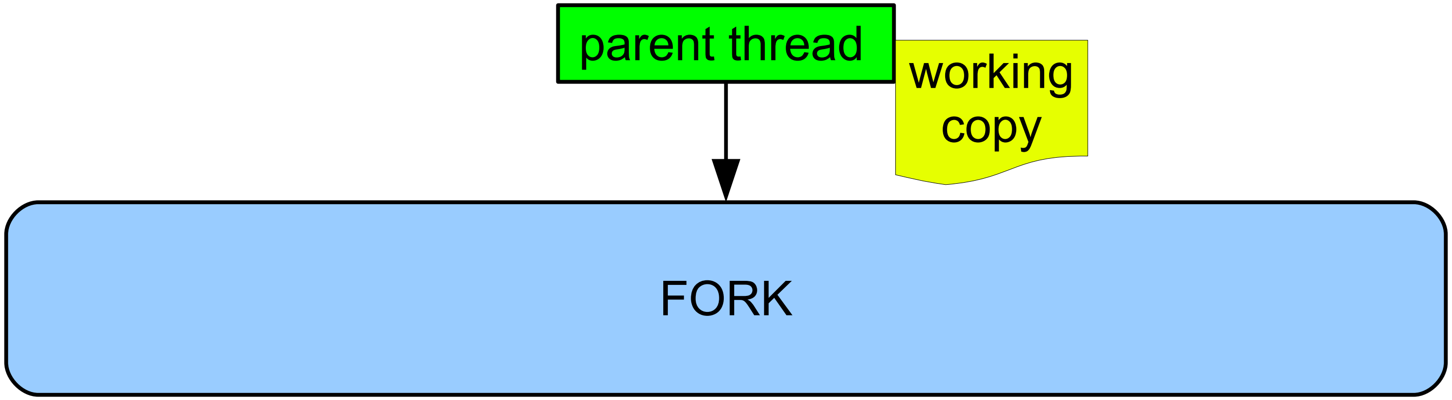
Var *x* gets its value each time from the previous *section* and then is thread-private

- The Goal √
- Background & Related Work √
- From OpenMP to DOMP Semantics√
- DOMP Runtime
- Initial Results
- Conclusion

DOMP Runtime


parent thread

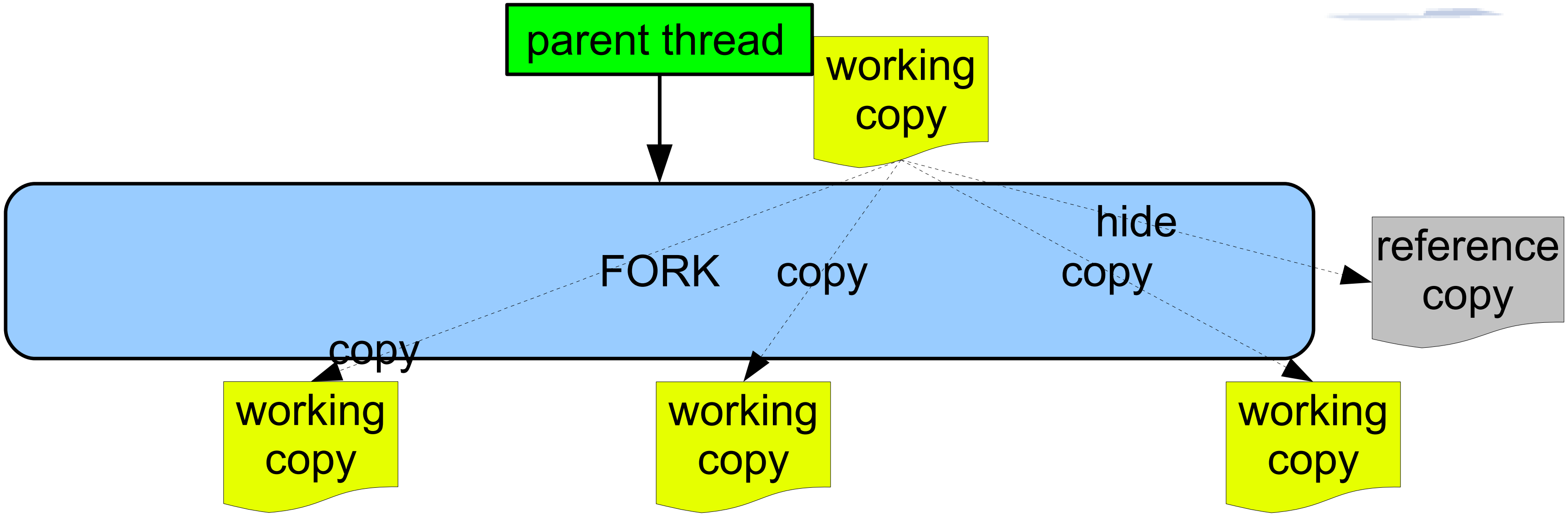
working copy

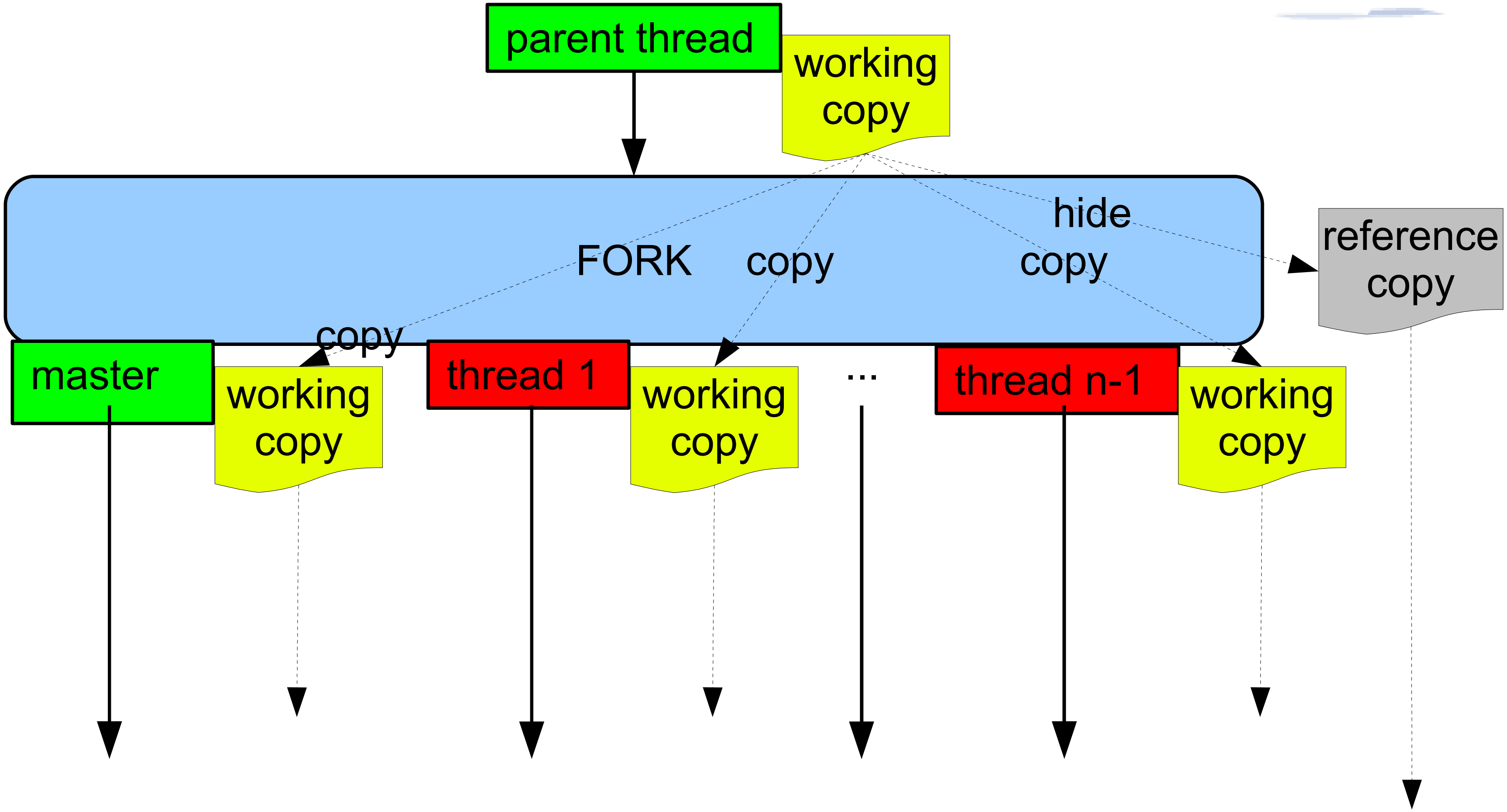


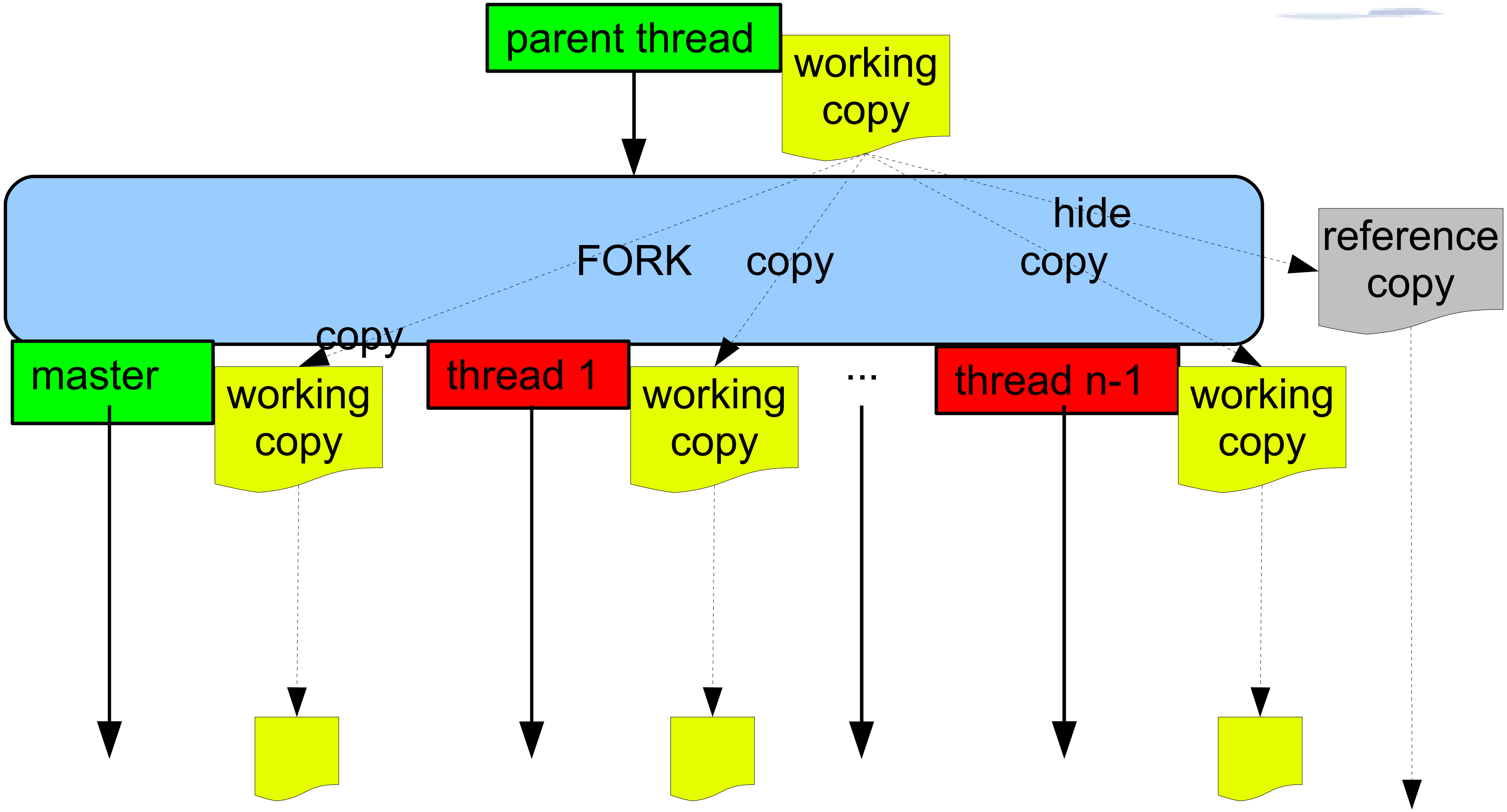
parent thread

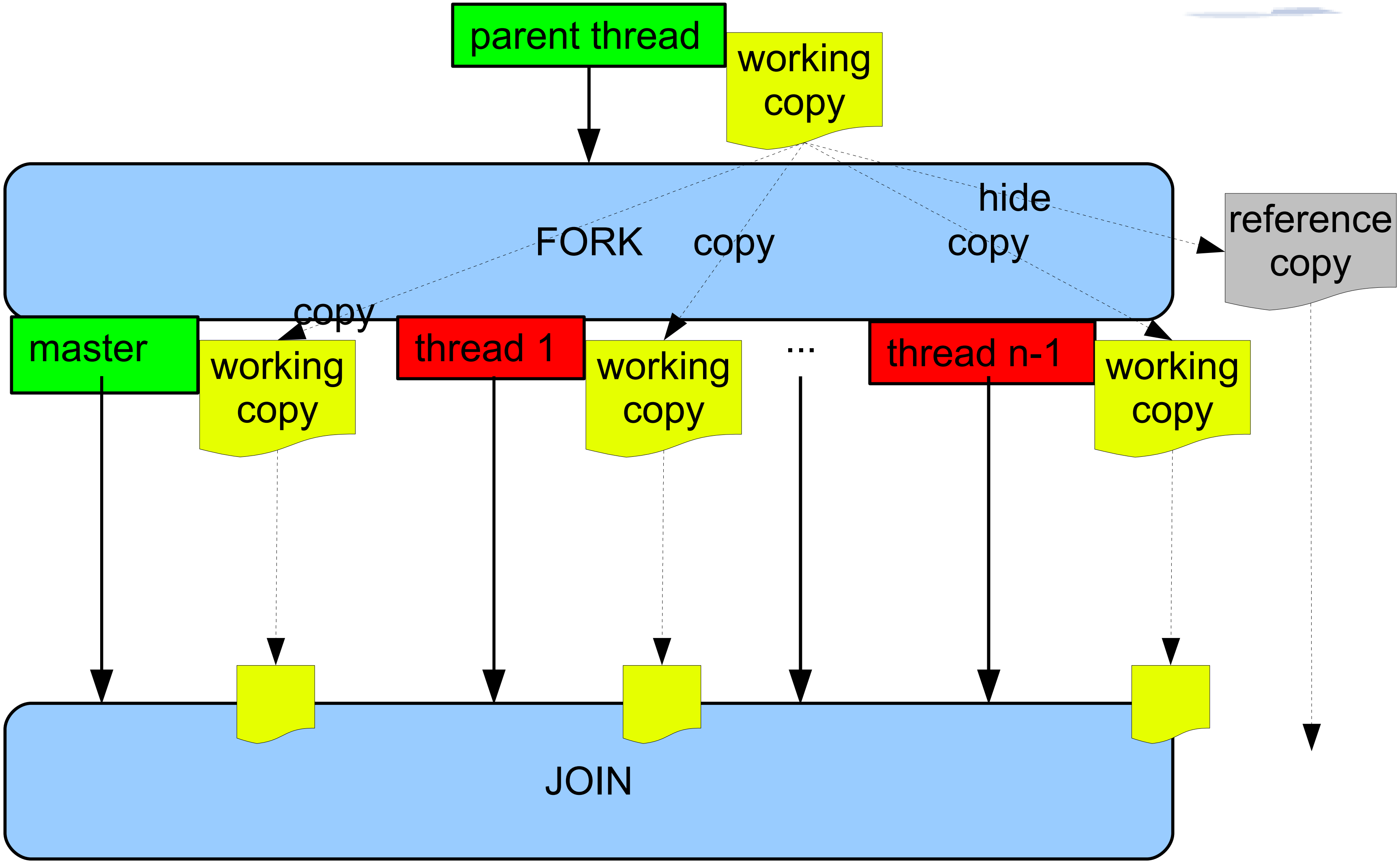
working copy

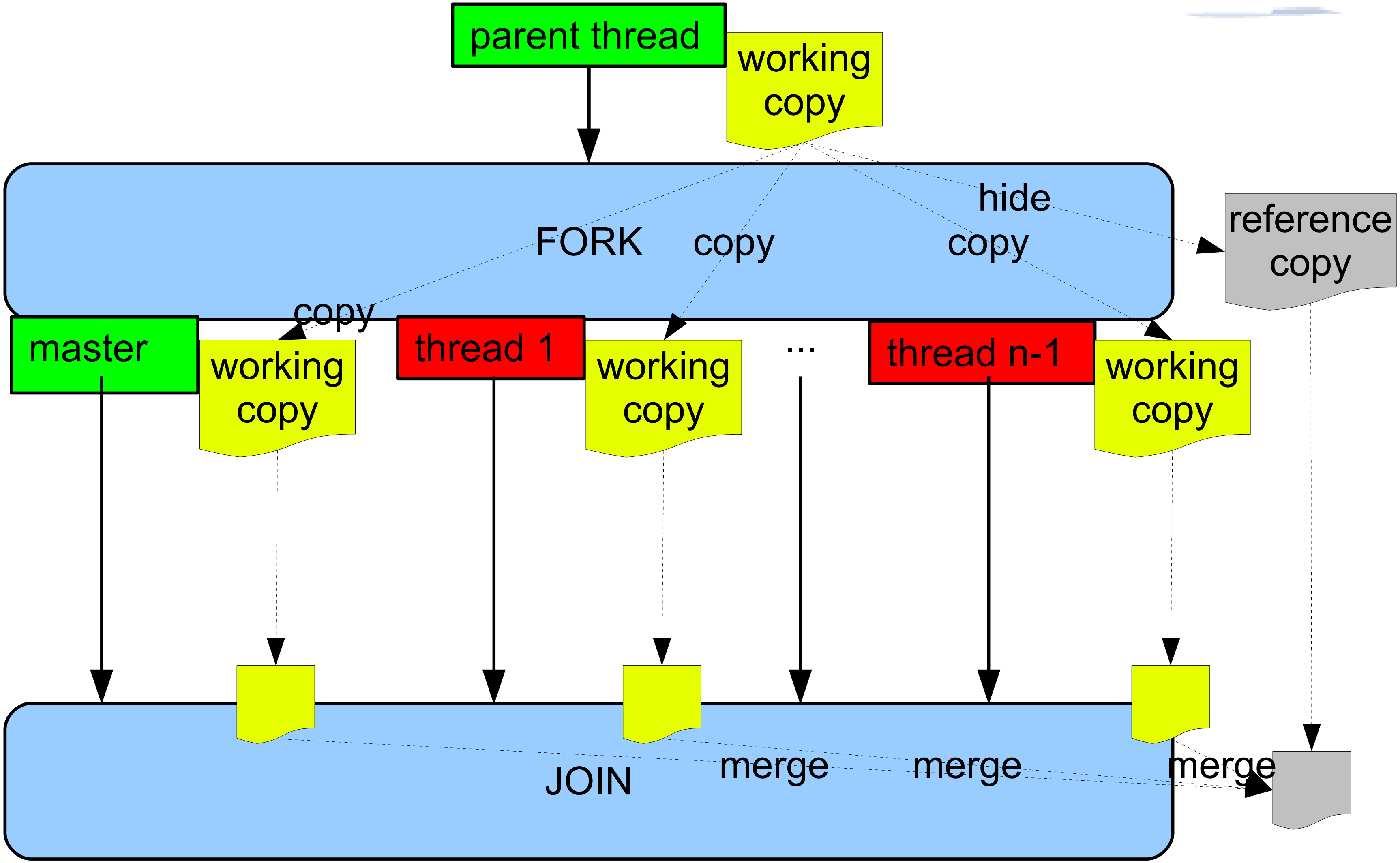
FORK

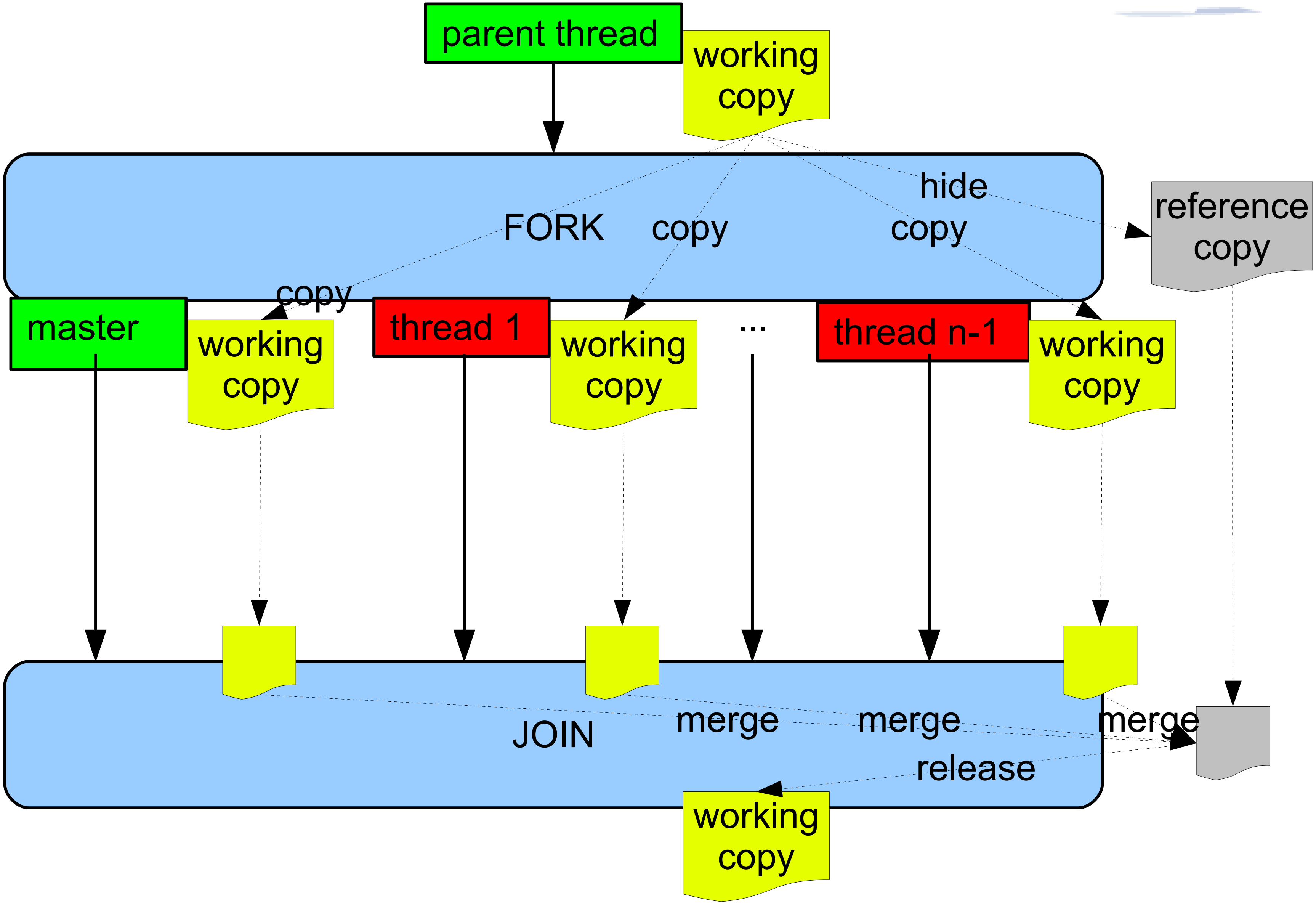

parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
working copy
working copy
working copy

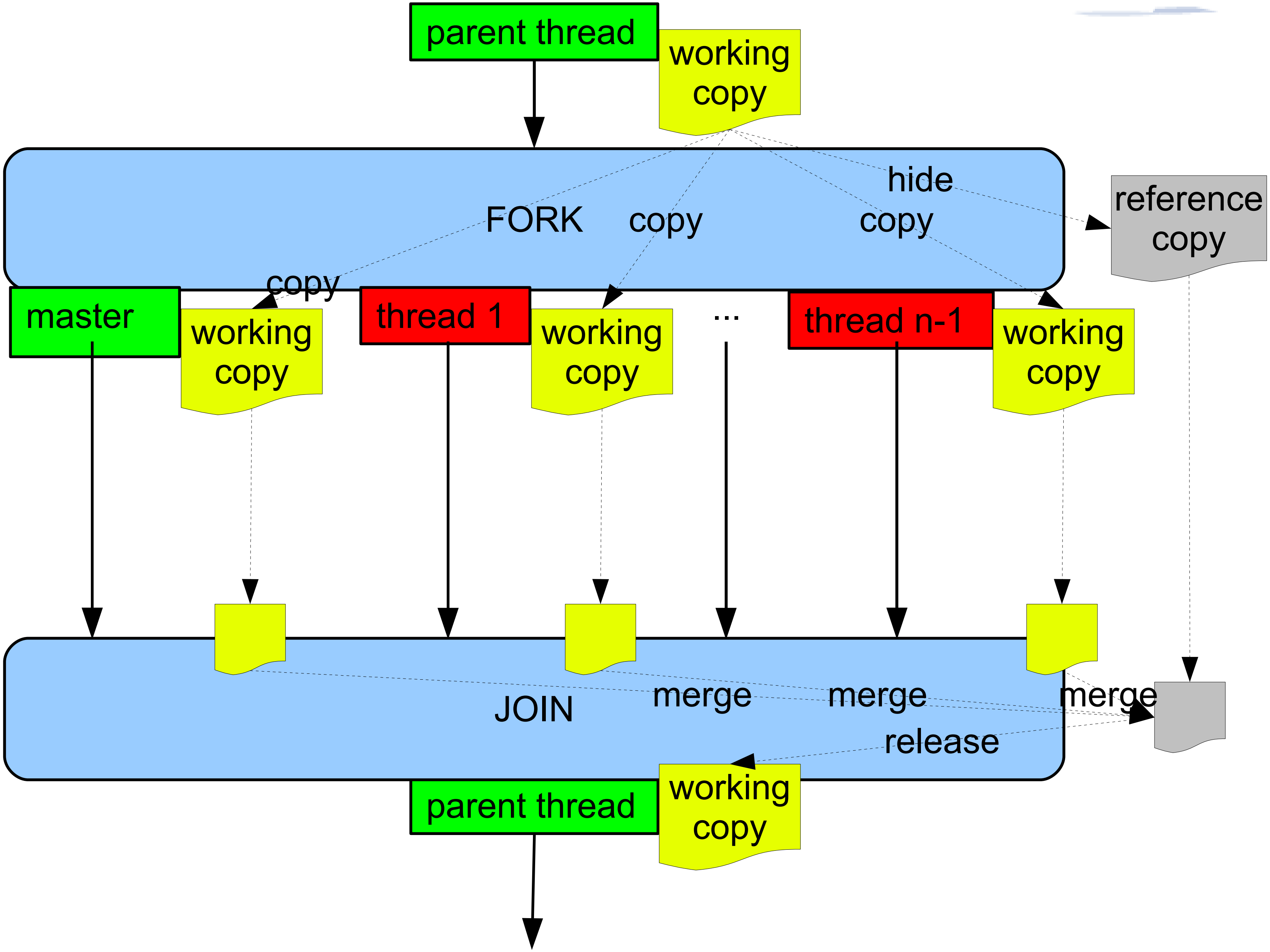

parent thread
working copy
hide
FORK
copy
copy
copy
reference copy
master
working copy
thread 1
working copy
...
thread n-1
working copy


parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
master
working copy
thread 1
working copy
…
thread n-1
working copy


parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
master
working copy
thread 1
working copy
...
thread n-1
working copy
JOIN


parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
master
working copy
thread 1
working copy
...
thread n-1
working copy
JOIN
merge
merge
merge


parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
master
working copy
thread 1
working copy
...
thread n-1
working copy
JOIN
merge
merge
merge
release
working copy


parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
master
working copy
thread 1
working copy
...
thread n-1
working copy
JOIN
merge
merge
merge
release
parent thread
working copy

- The Goal √
- Background & Related Work √
- From OpenMP to DOMP Semantics√
- DOMP Runtime √
- Efficiency
- Conclusion

Efficiency

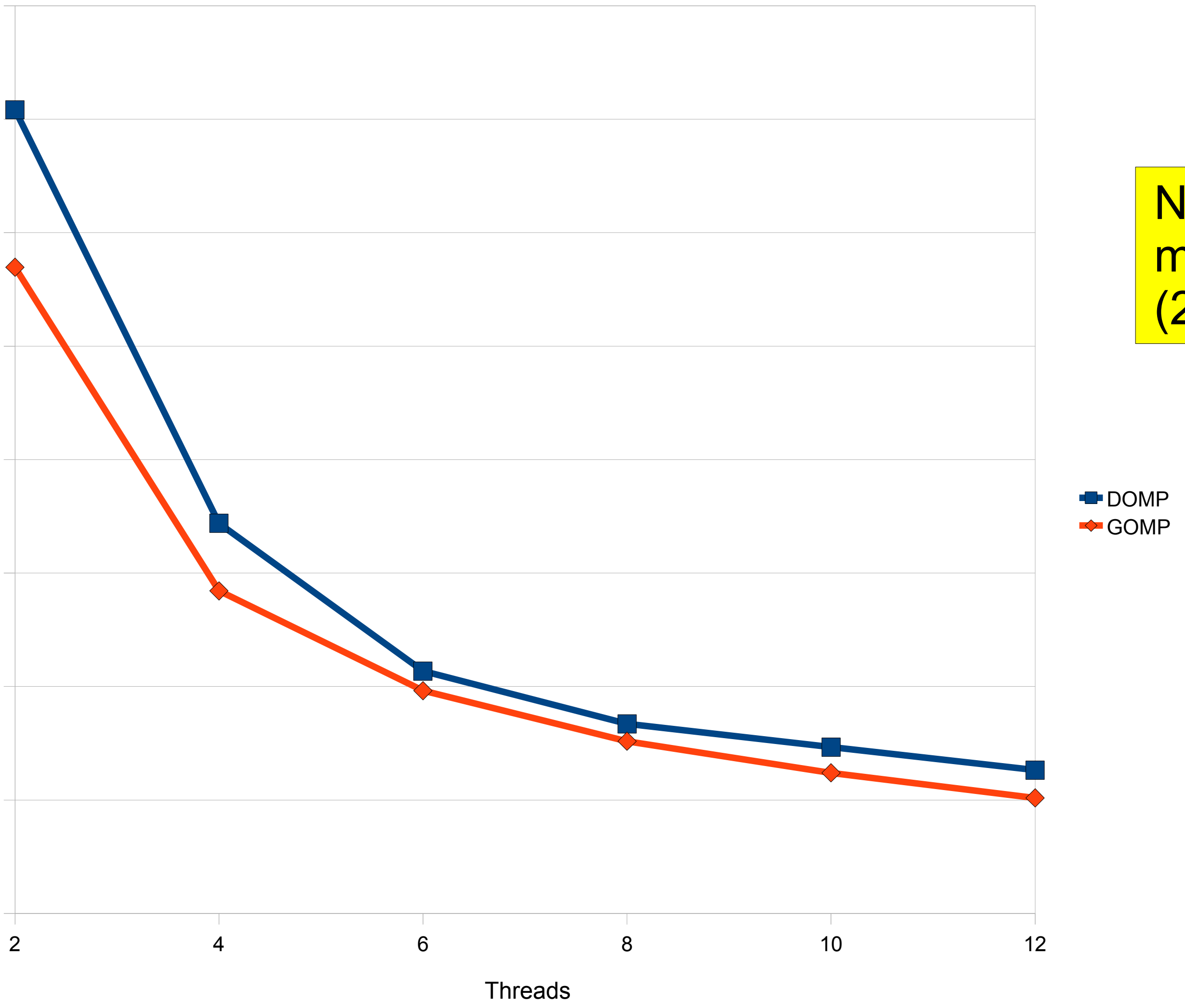
Can DOMP be Efficient?

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

Previous Experience

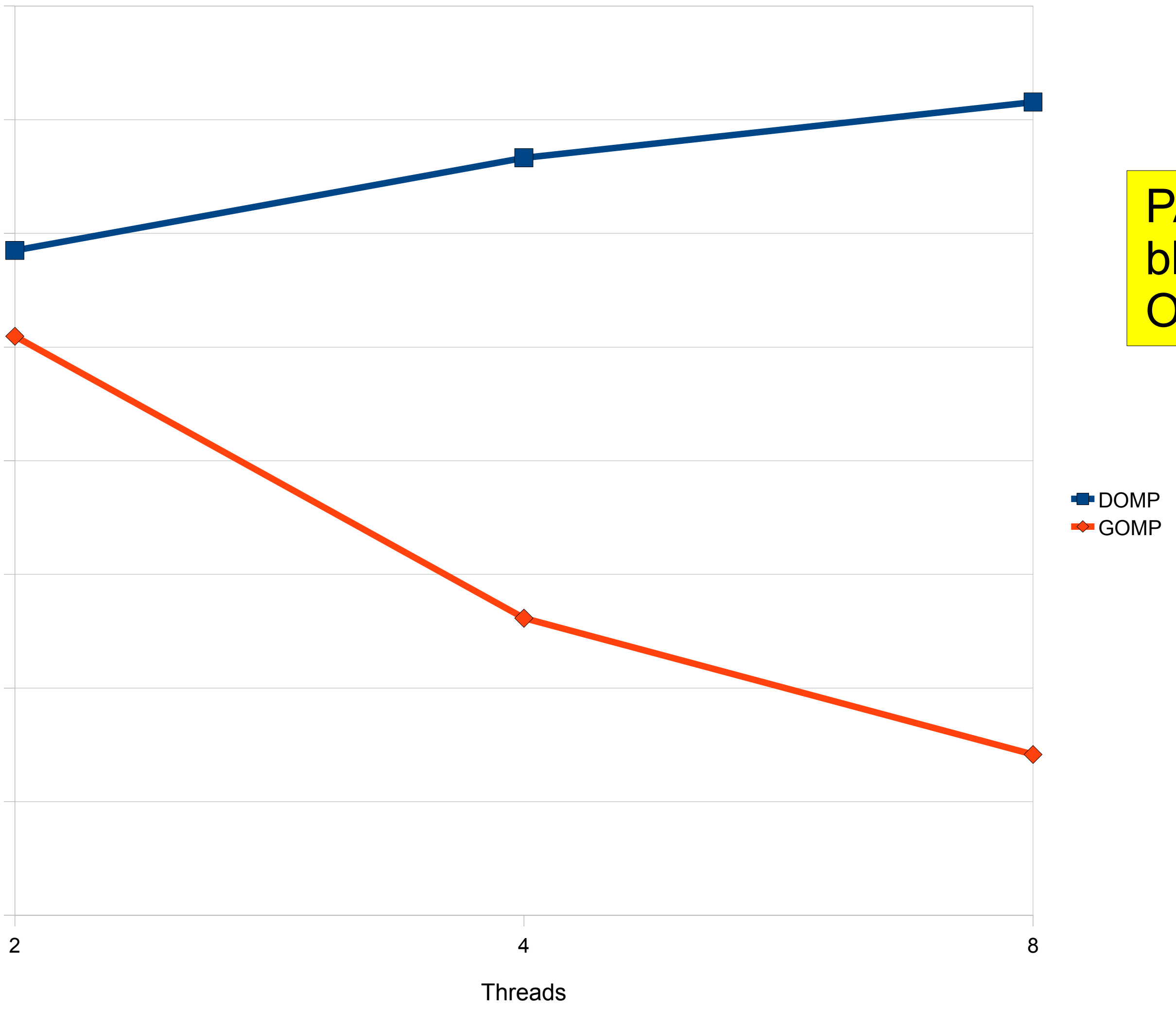
- *Dthreads*, *Determinator*—both naturally deterministic
- Good results for some benchmarks
- Great for “embarrassingly parallel” applications
- Fine-grained parallelism can be expensive

The Good News``



Naïve matrix multiplication (2048 x 2048)

The Not-So-Good News



PARSEC blackscholes for OpenMP

By the way ...

In testing, DOMP uncovered a hitherto unnoticed data race in the blackscholes (OpenMP version) code. ☺

- Bug-finding is not DOMP's primary goal
- But nice that its model shakes out concurrency bugs automatically

Conclusion

- DOMP: proposed race-free, deterministic parallel programming framework
- API based on OpenMP—new semantics & extensions
- Applies working-copies approach to enforce determinism
- Combines expressiveness, reliability, and (we hope) efficiency

Thank you