Deterministic OpenMP

Amittai 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

x = 1 | x = 2

x == 1 ?
x == 2

x = 0

x = 2 | y = x

y == 0 ?
x == 2 y == 2

x = 0

lock | lock
x++ | x*=2
unlock | unlock

x == 1 ?
x == 2

A	B
lock(x)	lock(x)
x := x + 2	x := x * 3
unlock(x)	unlock(x)

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

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

HEISENBUG

Accesses remained *unordered*

Determinism

- *program* : *input* → (*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.

Potentially useful but can be problematic

Run only deterministic programs.

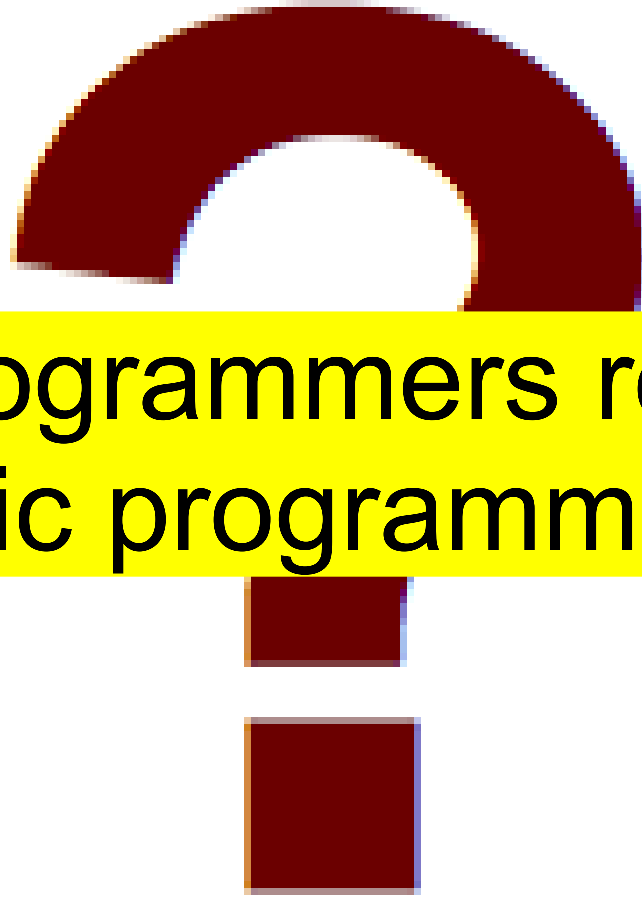
Enforce a deterministic programming model.

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



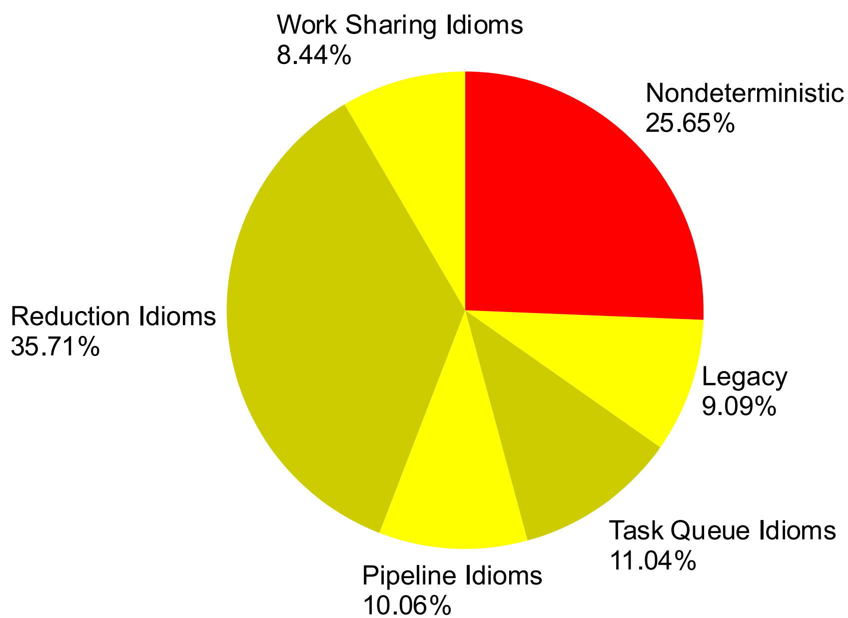
But can programmers really use a deterministic programming model?

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

SLOW

- OR -

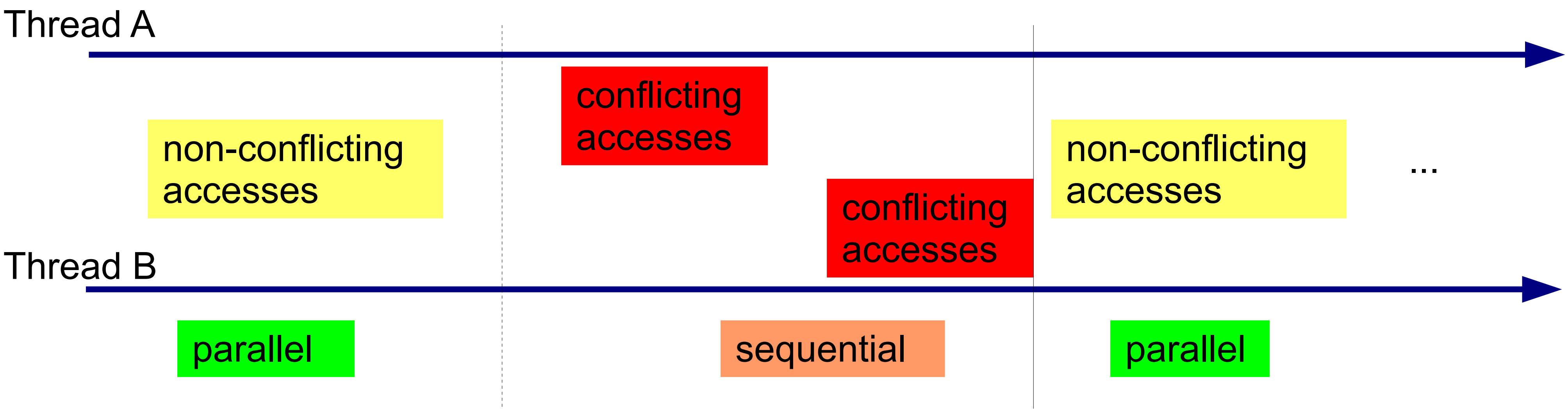
Require special

$$$

Deterministic Schedulers

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

Dedeterministic Scheduling



Schedule Dependency

```
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++;
}
```

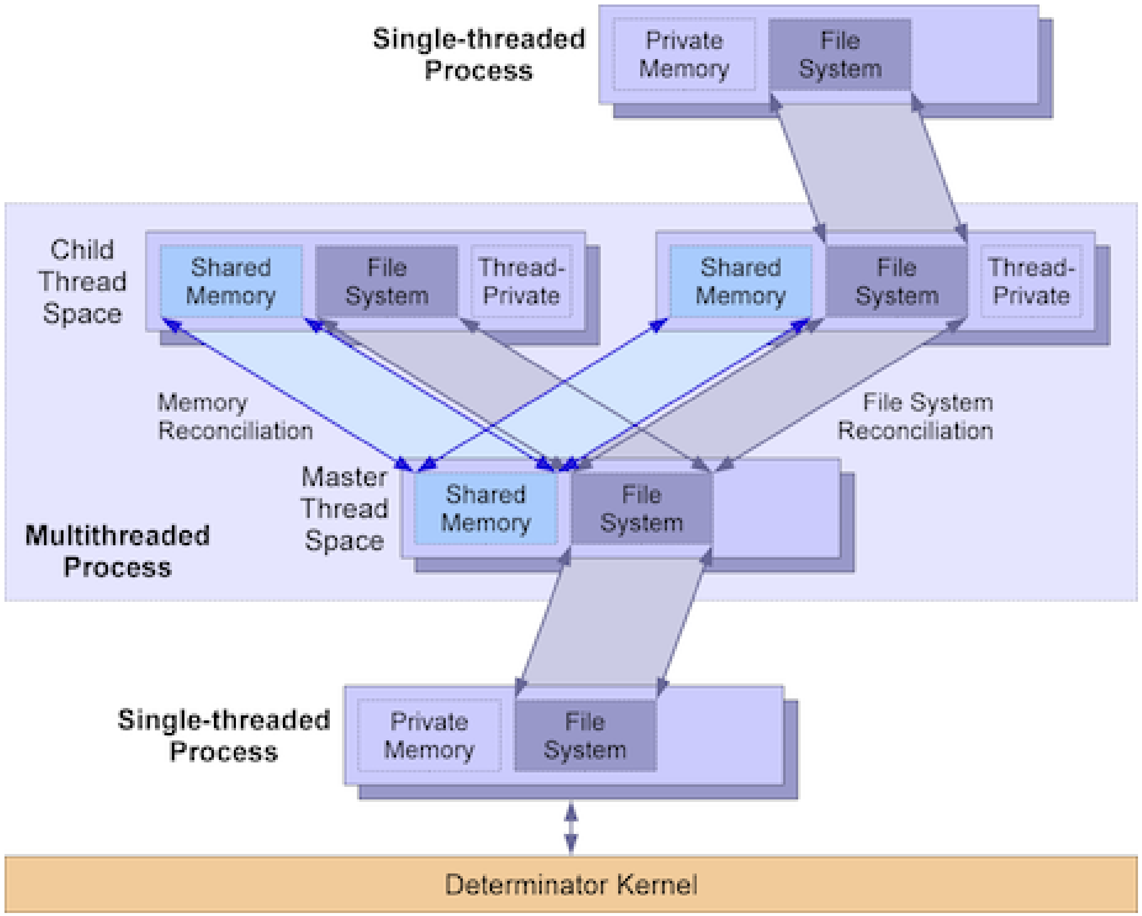
Thread A	Thread B	
$t_1 \leftarrow$ input_is_typical jump_zero t_1 L1 call do_a_lot	call do_a_little ret $t_2 \leftarrow x$	parallel Q_n
... ... ret	add t_2 1 $x \leftarrow t_2$	parallel Q_{n+1}
L1: $t_1 \leftarrow x$ add t_1 1 $x \leftarrow t_1$ ...		parallel Q_{n+2}

Schedule Dependency

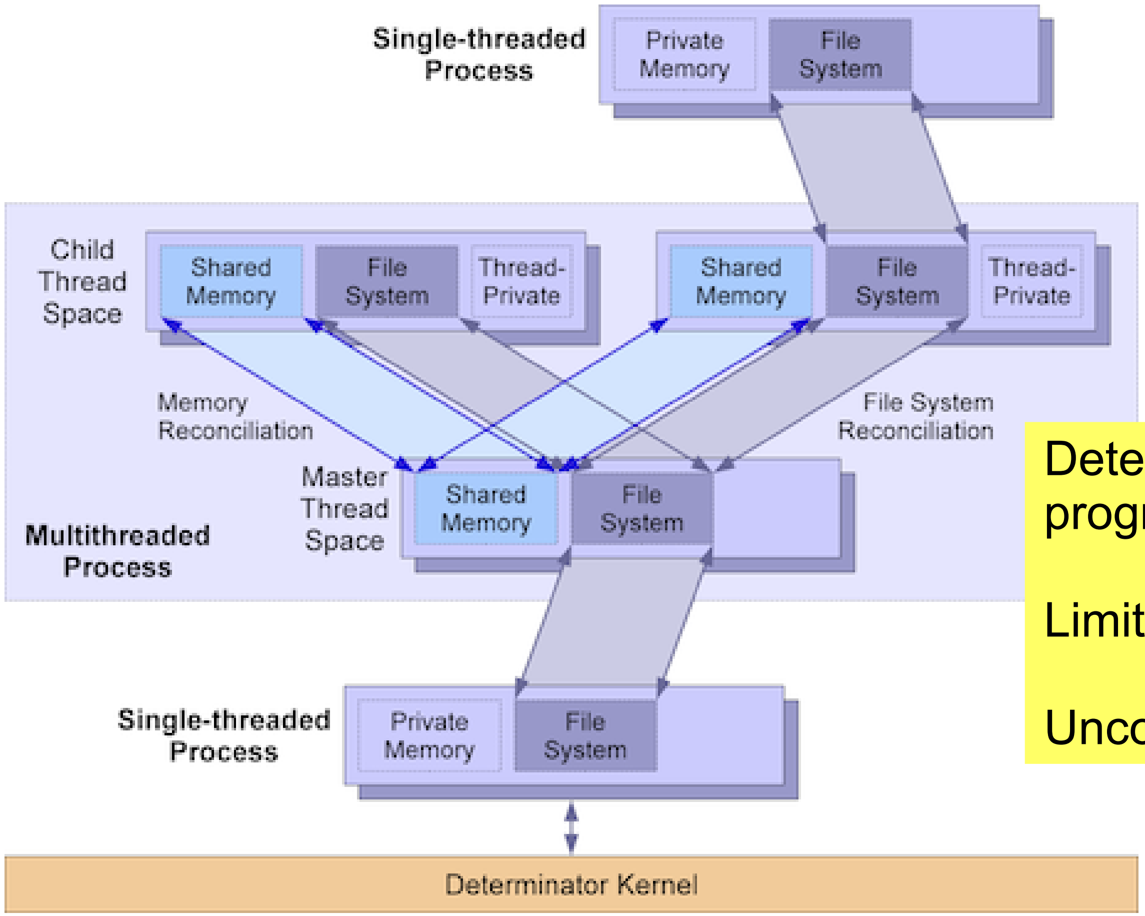
```
x = 42;
// Thread A:
{
  if (input_is_typical)
    do_a_lot();
  x++;
}
// Thread B:
{
  do_a_little();
  x++;
}
```

Thread A	Thread B	
$t_1 \leftarrow$ input_is_typical jump_zero t_1 L1 call do_a_lot	call do_a_little ret $t_2 \leftarrow x$	parallel Q_n
ret L1: $t_1 \leftarrow x$ add t_1 1	add t_2 1	parallel Q_{n+1}
$x \leftarrow t_1$	$x \leftarrow t_2$	serial Q_{n+2}

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 *non*determinstic 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

```
long ProcessId;

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

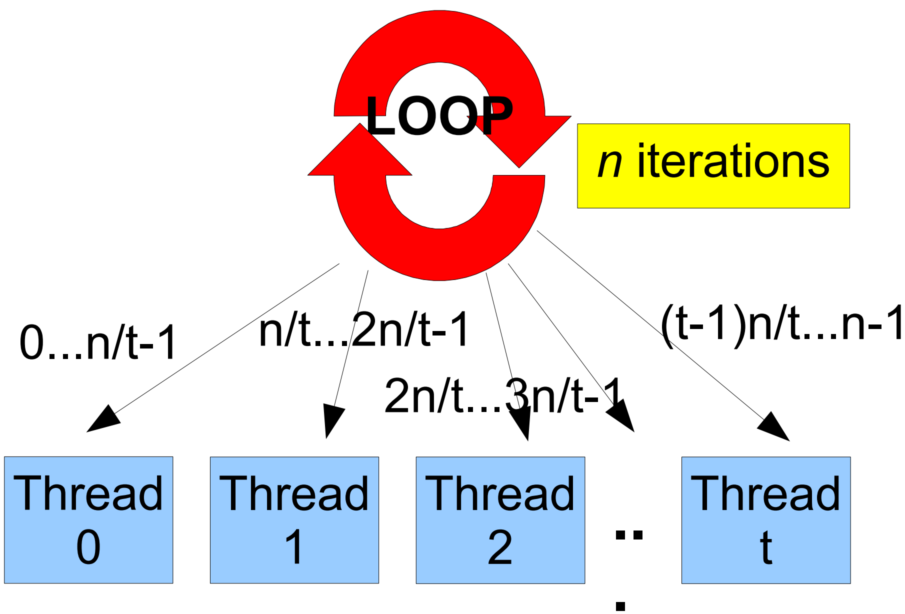
barnes (SPLASH2)

Work sharing

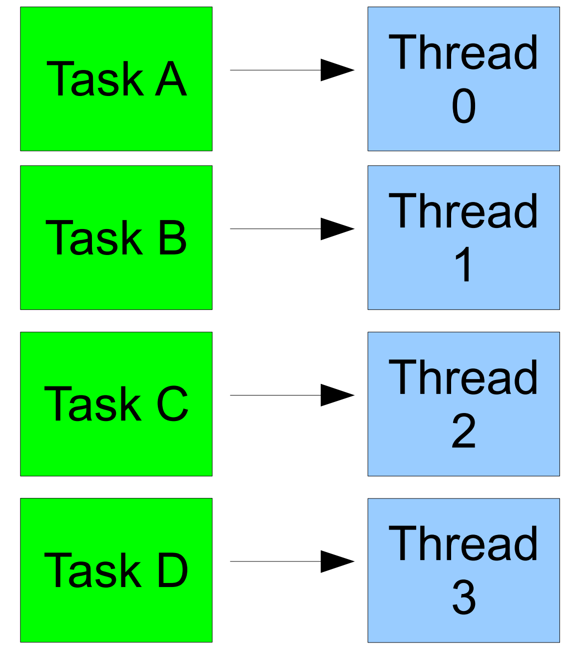
Idioms

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

Work Sharing

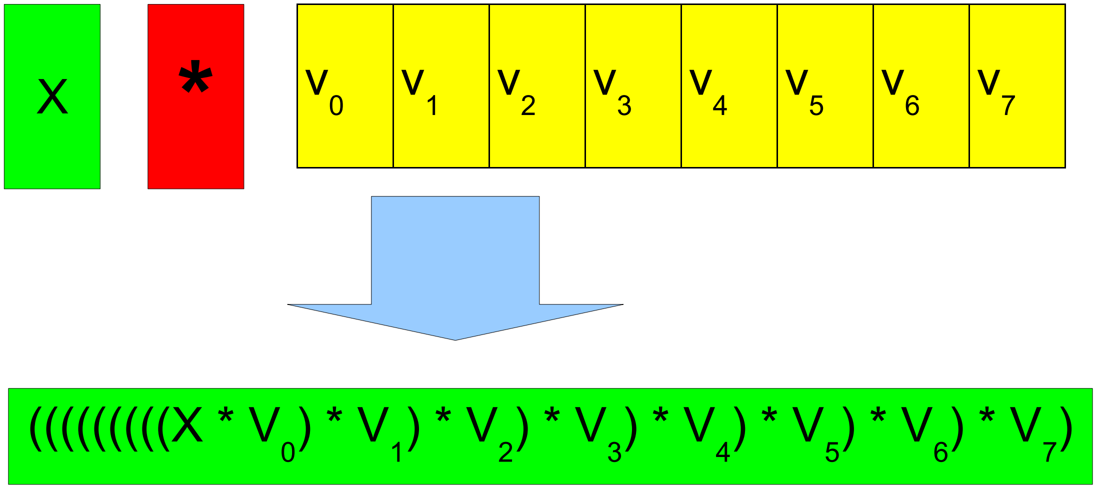


“Data Parallelism”
cf. OpenMP LOOP work sharing construct

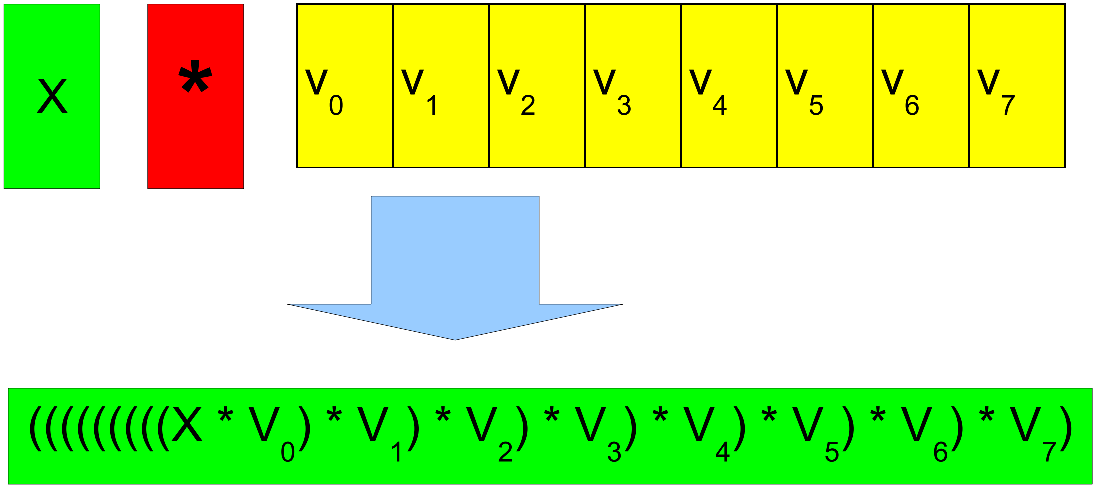


“Task Parallelism”
cf. OpenMP `sections` and `task` work sharing constructs

Reduction



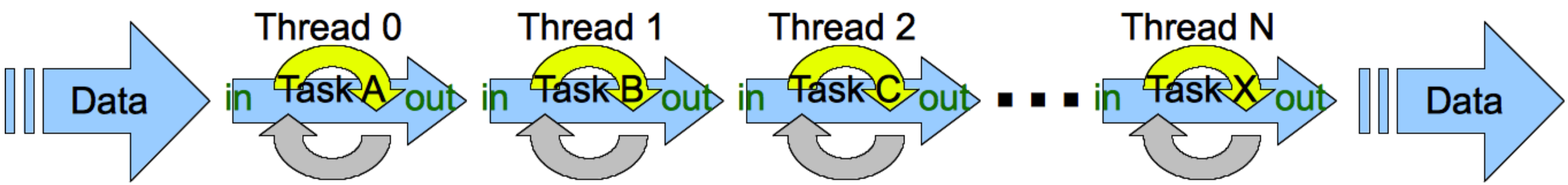
Reduction



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

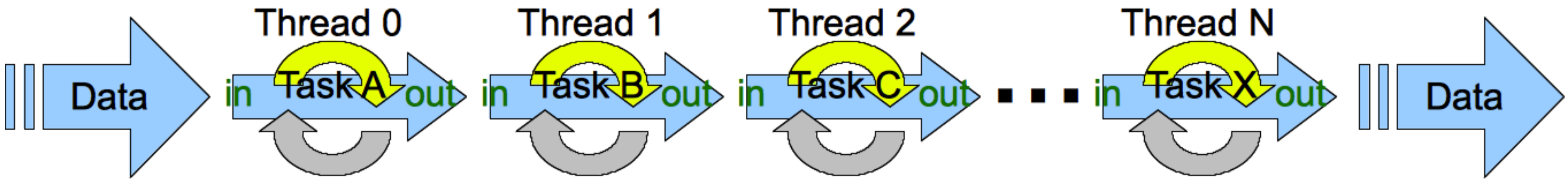
OpenMP's reduction construct allows only scalar types and simple operations.

Pipeline

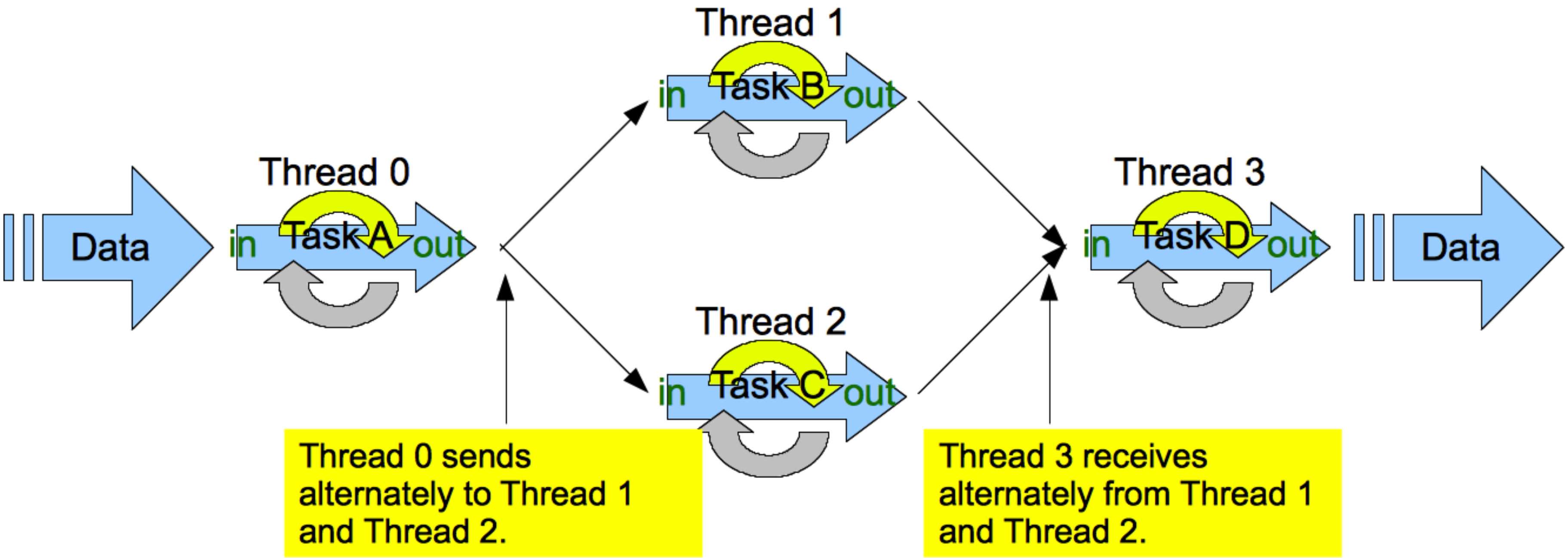


Thread 0
Thread 1
Thread 2
Thread N
Data
in
Task A
out
in
Task B
out
in
Task C
out
in
Task X
out
Data

Pipeline

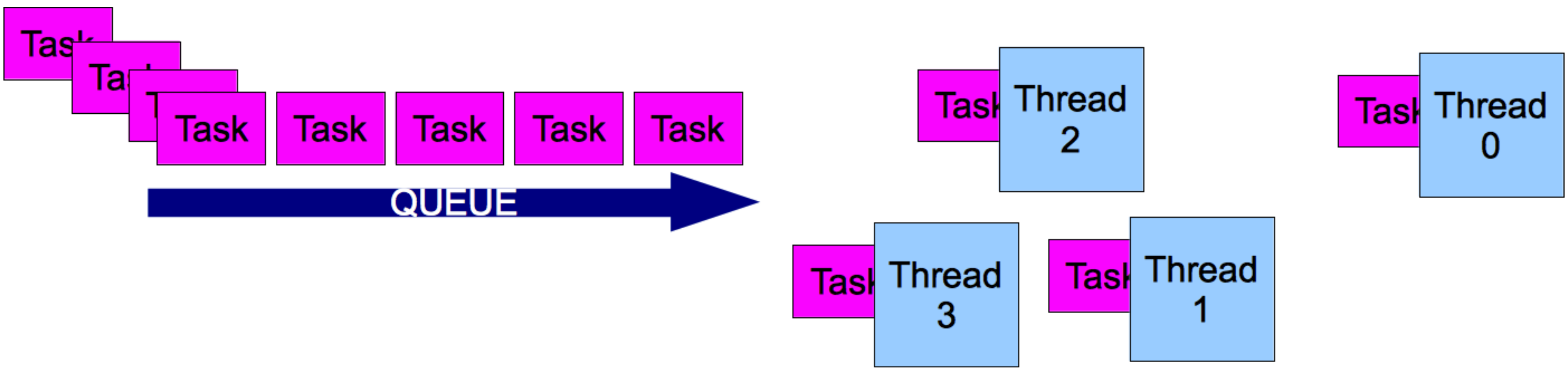


Thread 0
Thread 1
Thread 2
Thread N
Data
in
Task A
out
in
Task B
out
in
Task C
out
in
Task X
out
Data



Thread 1
in
Task B
out
Thread 0
Data
in
Task A
out
Thread 3
in
Task D
out
Data
Thread 2
in
Task C
out
Thread 0 sends alternately to Thread 1 and Thread 2.
Thread 3 receives alternately from Thread 1 and Thread 2.

Task Queue



Task
Task
Task
Task
Task
Task
QUEUE
Thread 2
Thread 0
Thread 3
Thread 1

Idioms

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

DETERMINISTIC IDIOMS

- Nondeterministic
  - Load balancing, random simulated interaction …

SPLASH2

		barnes	fmm	ocean	radiosity	raytrace	volrend	water-nsquared	water-spatial	cholesky	fft	lu	radix	TOTAL	
Deterministic Constructs	fork/join	1	2	1	3	1	5	1	1	1	1	2	1	20	7%
	barrier	6	13	40	5	1	15	9	9	4	7	10	7	126	46%
	work sharing	-	-	-	-	-	-	-	-	-	-	-	-	0	0%
	reduction	-	-	-	-	-	-	-	-	-	-	-	-	0	0%
Deterministic Idioms	work sharing	2	1	1	2	5	5	1	1	1	1	2	1	23	8%
	reduction	1	1	3	5	-	-	7	4	-	-	-	-	21	8%
	pipeline	-	3	-	-	-	-	-	-	-	-	-	2	5	2%
	task queue	-	-	-	7	-	-	-	-	2	-	-	-	9	3%
	legacy	1	15	-	-	6	1	-	1	4	-	-	-	28	10%
nondeterministic		2	8	-	23	2	6	-	2	-	-	-	-	43	16%

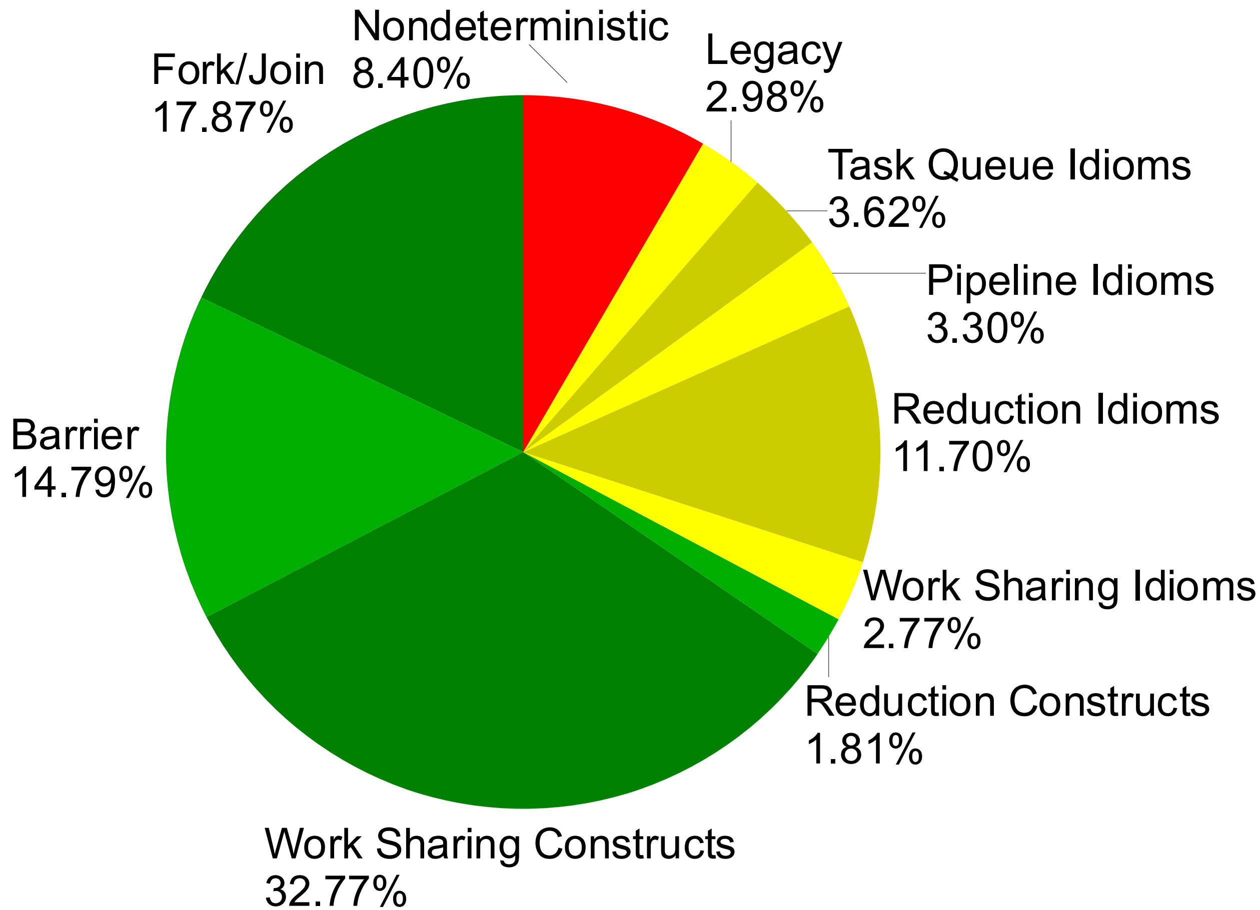
NPB-OMP

		BT	CG	DC	EP	FT	IS	LU	MG	SP	UA	TOTAL	
Deterministic Constructs	fork/join	12	7	1	3	8	7	12	11	13	60	134	25%
	barrier	-	8	-	-	-	1	4	-	-	-	13	2%
	work sharing	37	20	-	1	8	11	71	16	38	78	280	52%
	reduction	-	6	-	1	1	-	3	2	-	4	17	3%
Deterministic Idioms	work sharing	-	-	-	-	-	-	-	-	-	-	0	0%
	reduction	2	-	1	1	-	1	2	-	2	80	89	17%
	pipeline	-	-	-	-	-	-	5	-	-	-	5	1%
	task queue	-	-	-	-	-	-	-	-	-	-	0	0%
	legacy	-	-	-	-	-	-	-	-	-	-	0	0%
nondeterministic		-	-	-	-	-	-	-	-	-	-	0	0%
												538	

PARSEC

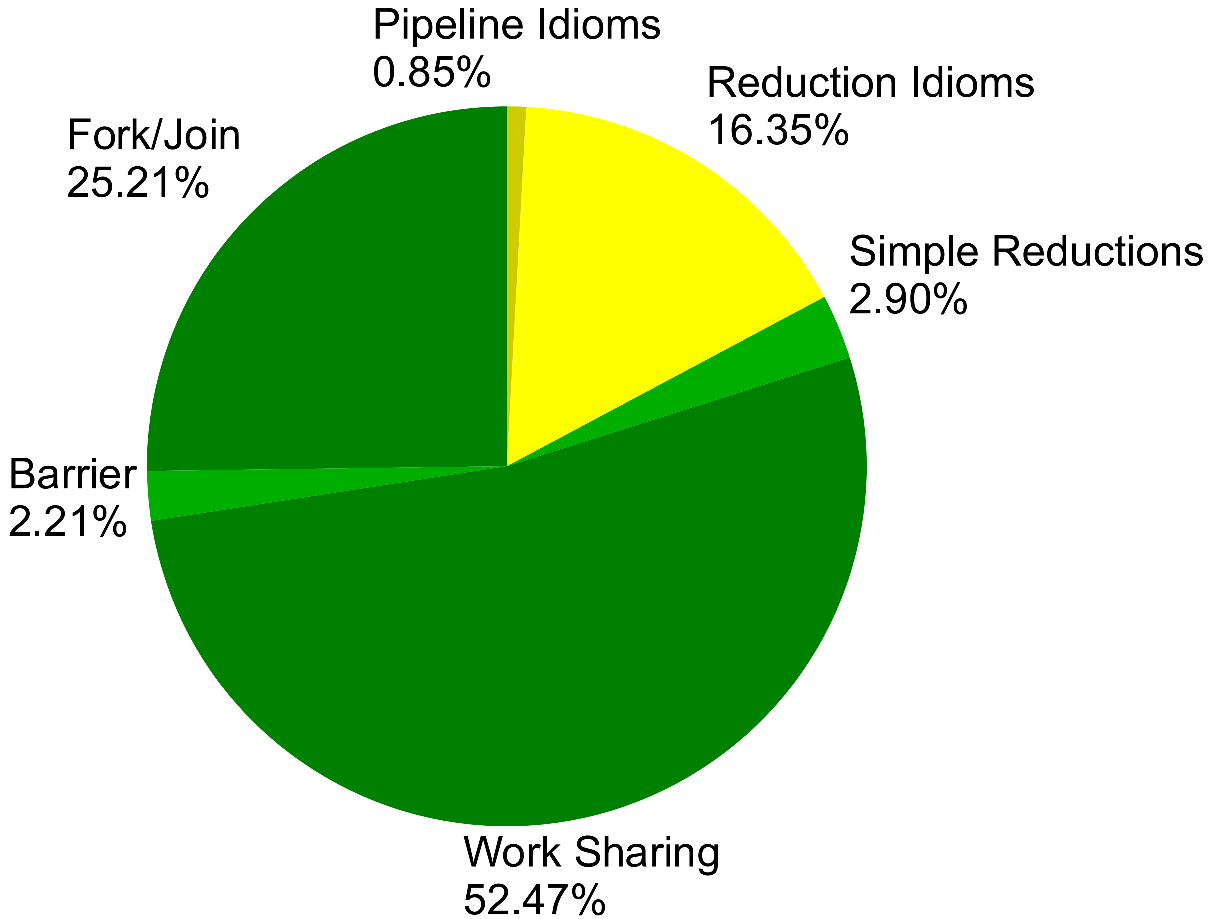
		blackscholes	bodytrack	facesim	ferret	fluidanimate	freqmine	raytrace	swaptions	vips	x264	canneal	dedup	streamcluster	TOTAL	
Deterministic Constructs	fork/join	2	5	2	1	13	7	1	3	1	2	1	5	5	48	23%
	barrier	-	-	-	-	14	-	3	-	-	-	1	-	34	52	25%
	work sharing	2	5	-	-	-	21	-	-	-	-	-	-	-	28	14%
	reduction	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0%
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

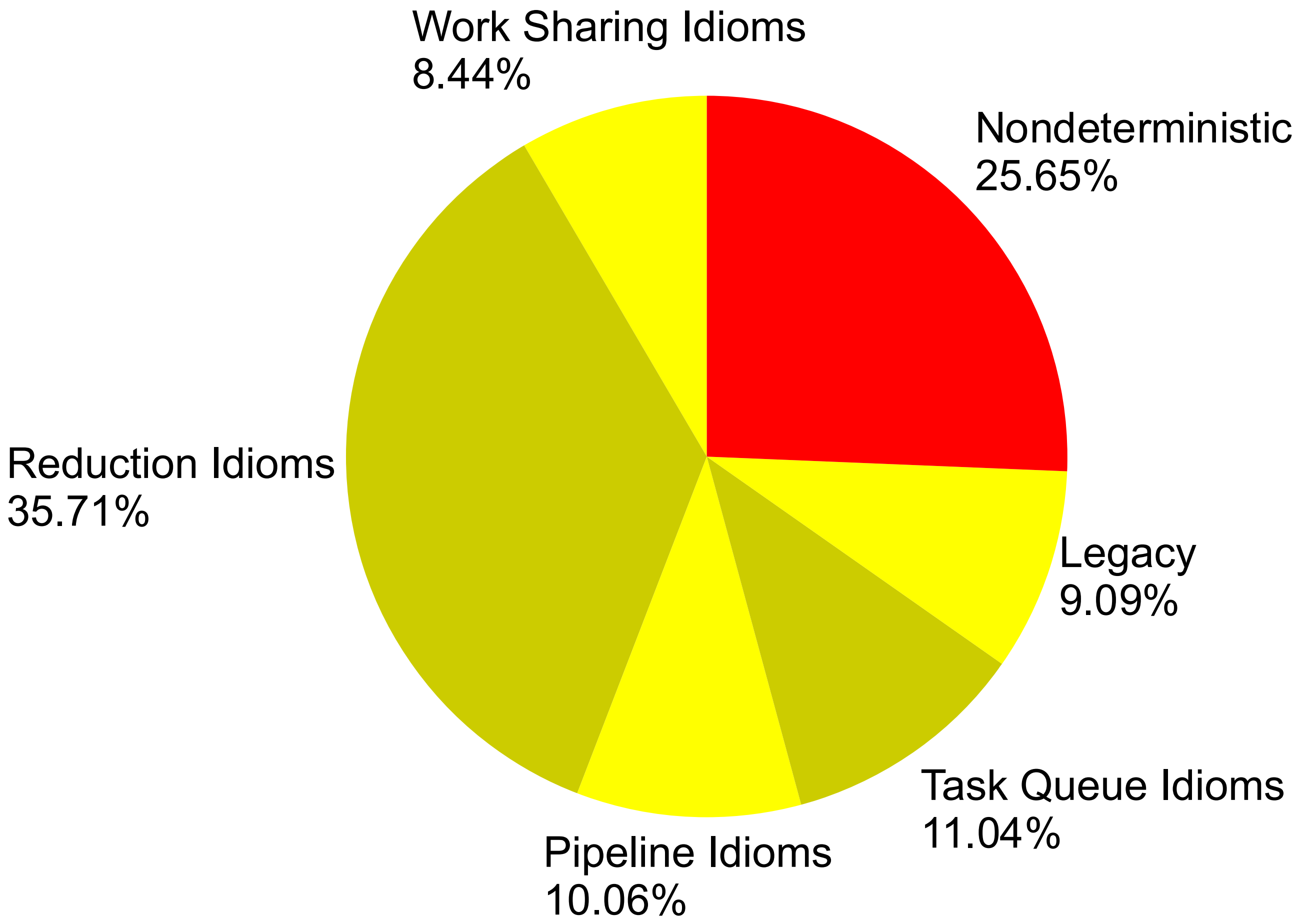


Nondeterministic
8.40%
Legacy
2.98%
Task Queue Idioms
3.62%
Pipeline Idioms
3.30%
Reduction Idioms
11.70%
Work Sharing Idioms
2.77%
Reduction Constructs
1.81%
Work Sharing Constructs
32.77%
Barrier
14.79%
Fork/Join
17.87%

OpenMP Benchmarks



Nondeterministic Synchronization



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

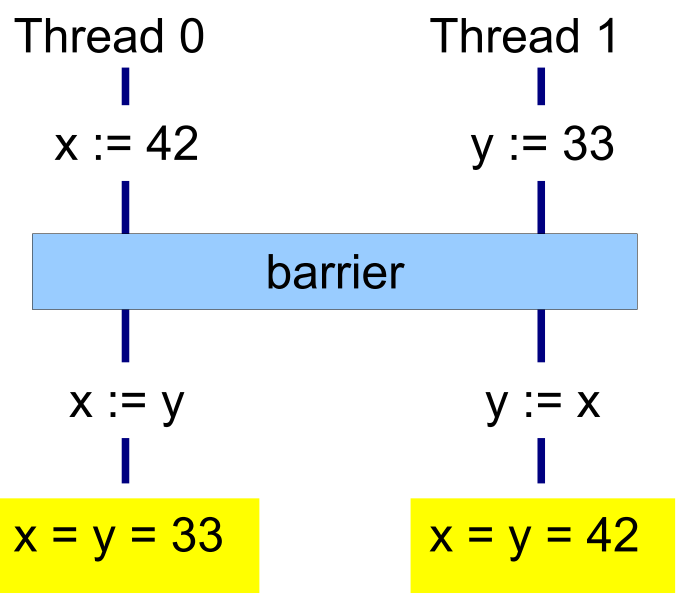
- The Big Picture √
- Background √
- Analysis √
- Design and Semantics
- Extended Reduction
- Implementation
- Evaluation
- Conclusion

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)



Memory Consistency Model Communication Events

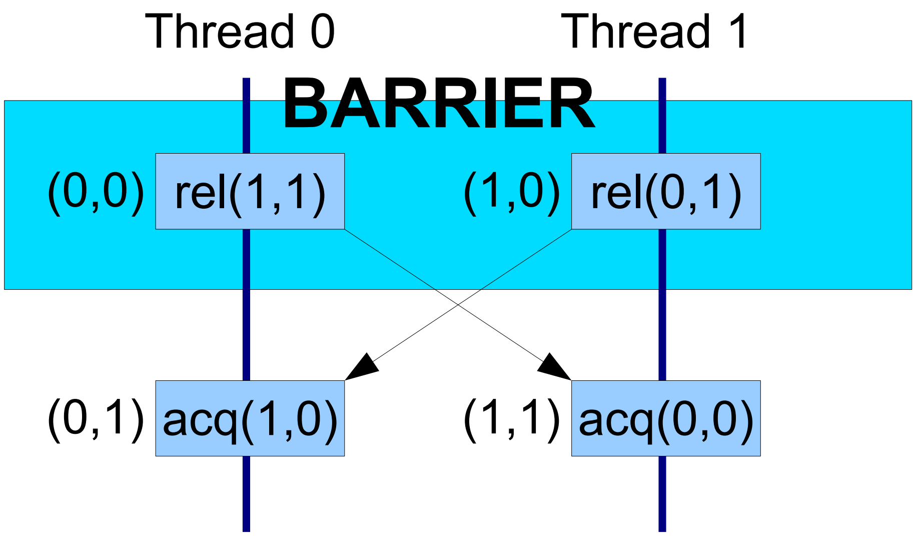
- Acquire
  - Acquires access to a location in shared memory
  - Involves a read
- Release
  - Enables access to a location in shared memory for other threads
  - Involves a write

Workspace Consistency

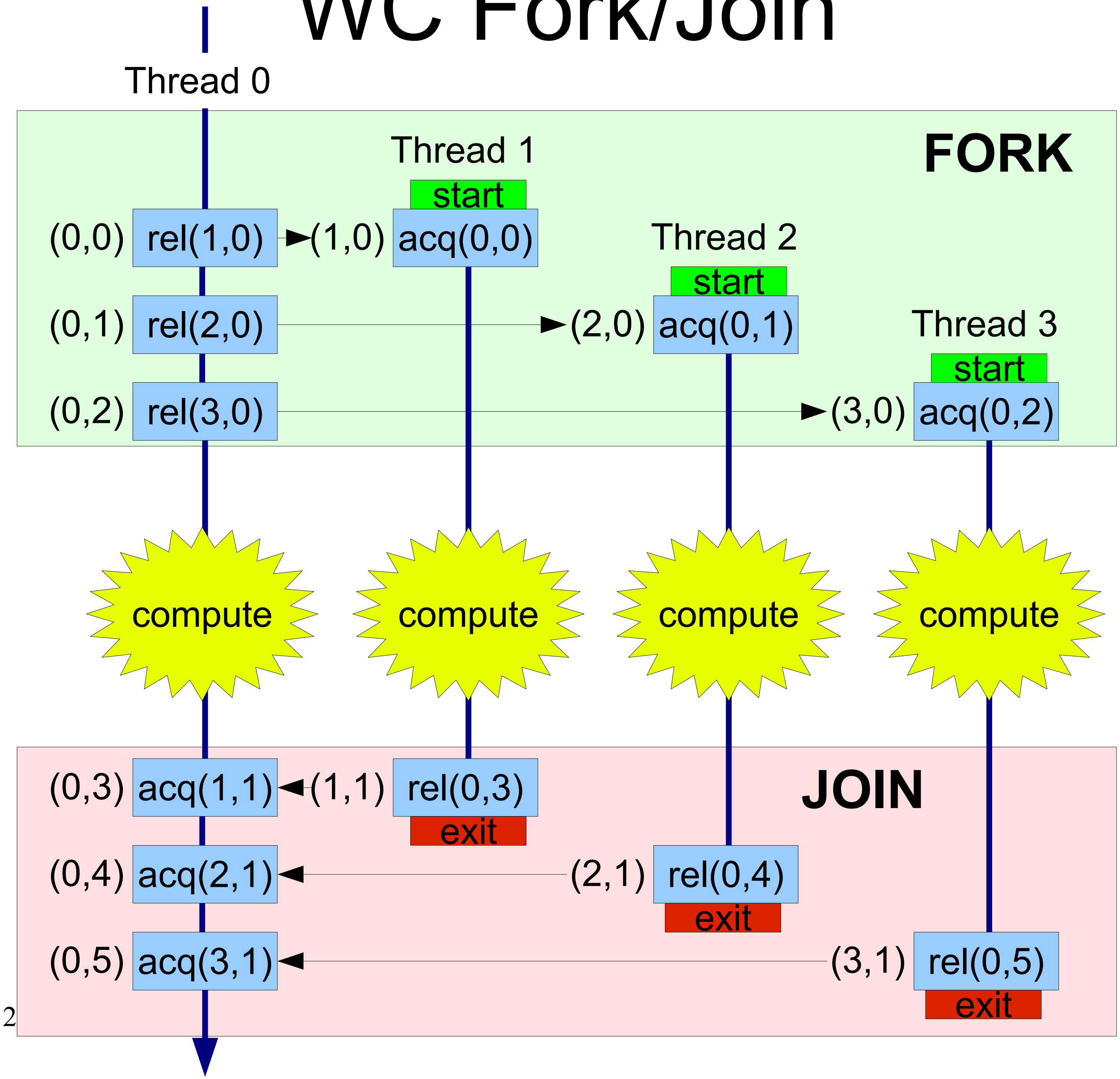
WoDet '11

- Pair each release with a determinate acquire
- Delay visibility of updates until the next synchronization event

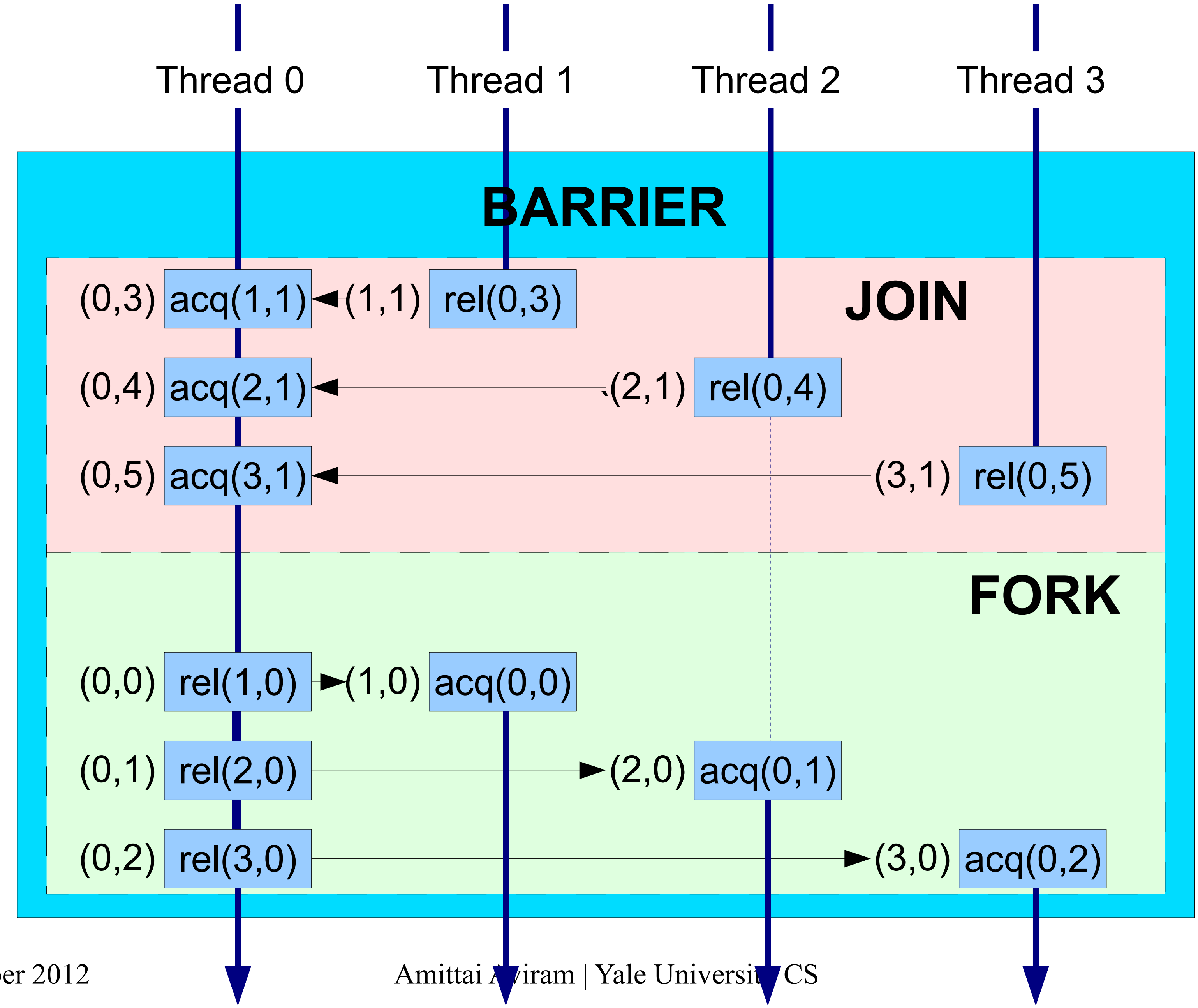
WC "Parallel Swap"



WC Fork/Join


Thread 0
FORK
Thread 1
start
(0,0) rel(1,0)
(1,0) acq(0,0)
Thread 2
start
(0,1) rel(2,0)
(2,0) acq(0,1)
Thread 3
start
(0,2) rel(3,0)
(3,0) acq(0,2)
compute
compute
compute
compute
JOIN
(0,3) acq(1,1)
(1,1) rel(0,3)
exit
(0,4) acq(2,1)
(2,1) rel(0,4)
exit
(0,5) acq(3,1)
(3,1) rel(0,5)
exit

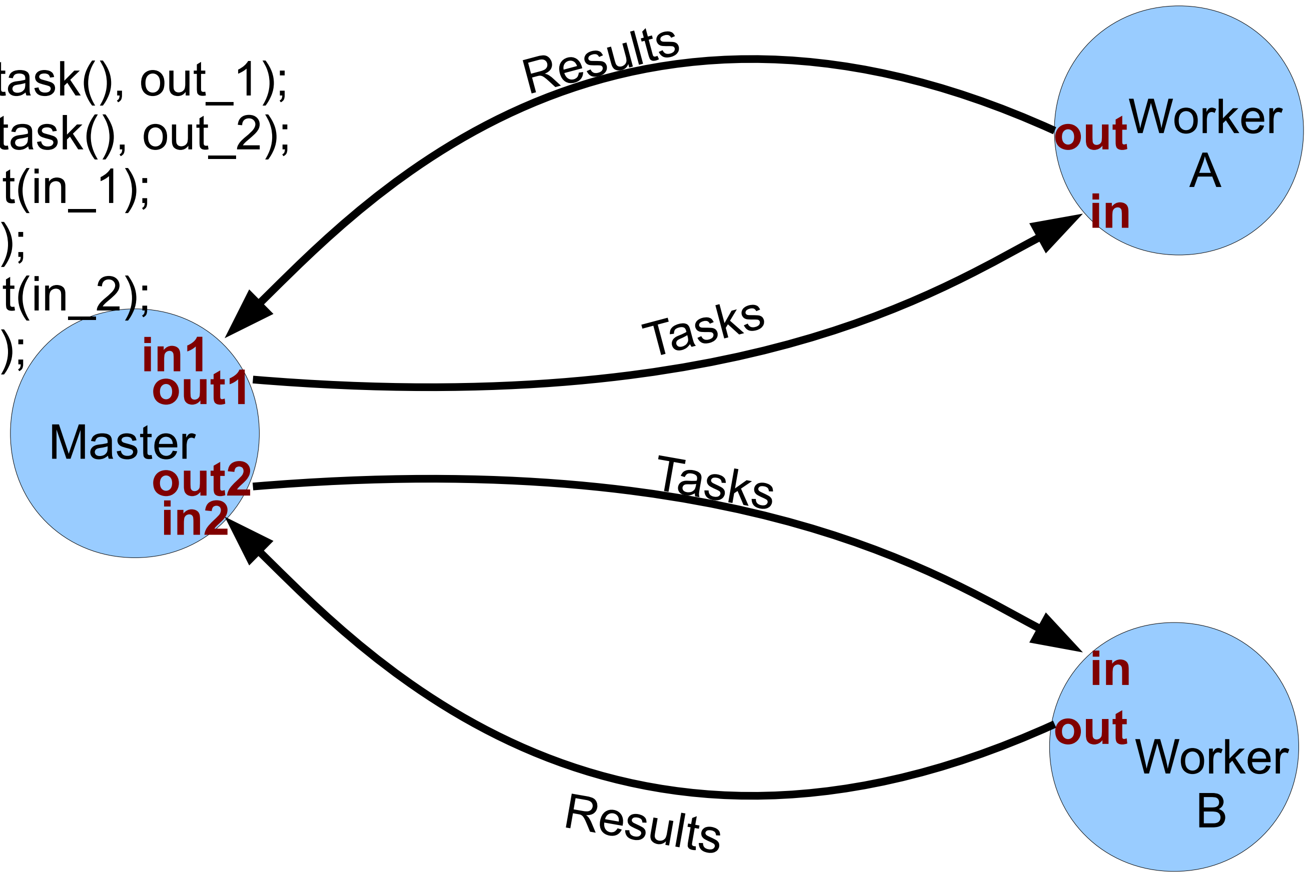
WC Barrier



Thread 0
Thread 1
Thread 2
Thread 3
BARRIER
JOIN
(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)
FORK
(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)

Kahn Process Network

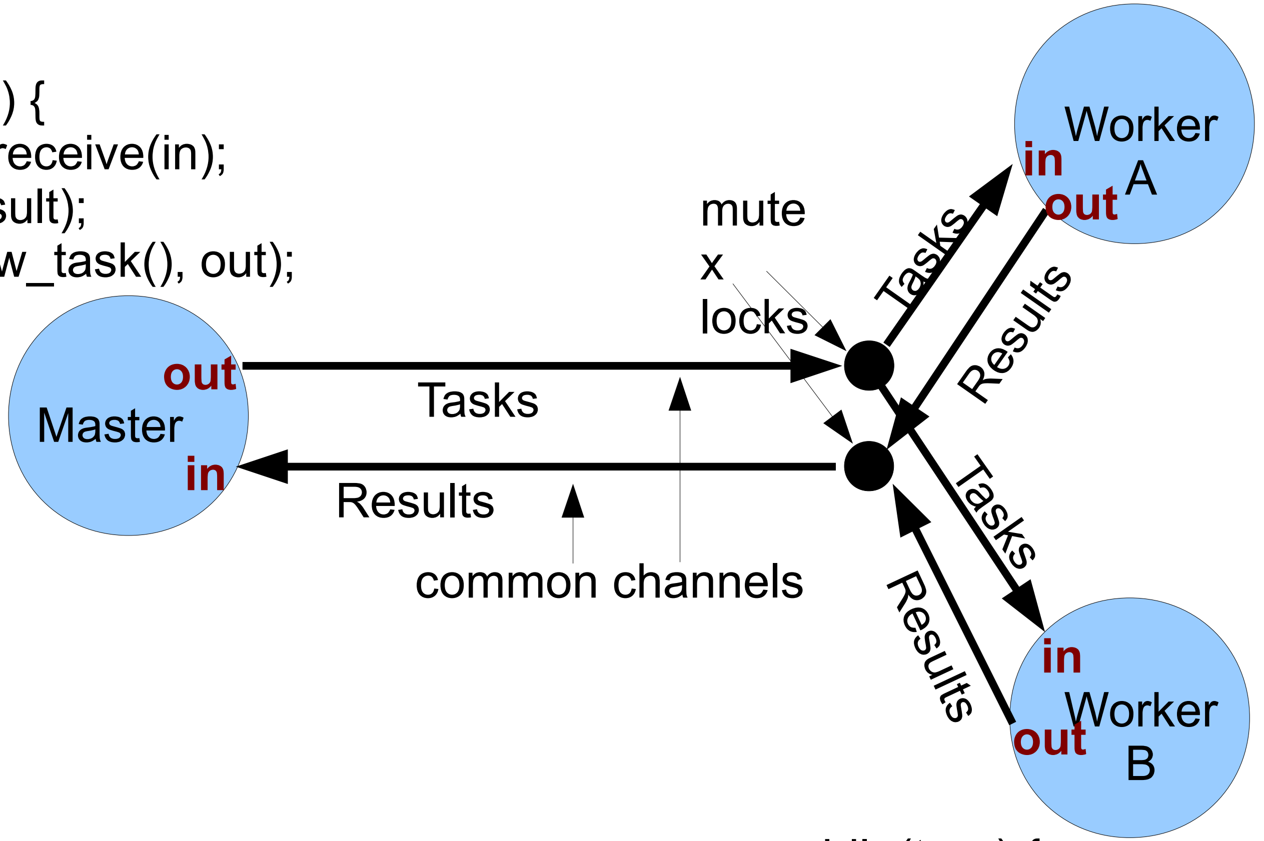
```
while (true) {
  send(new_task(), out_1);
  send(next_task(), out_2);
  result = wait(in_1);
  store(result);
  result = wait(in_2);
  store(result);
}
```




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

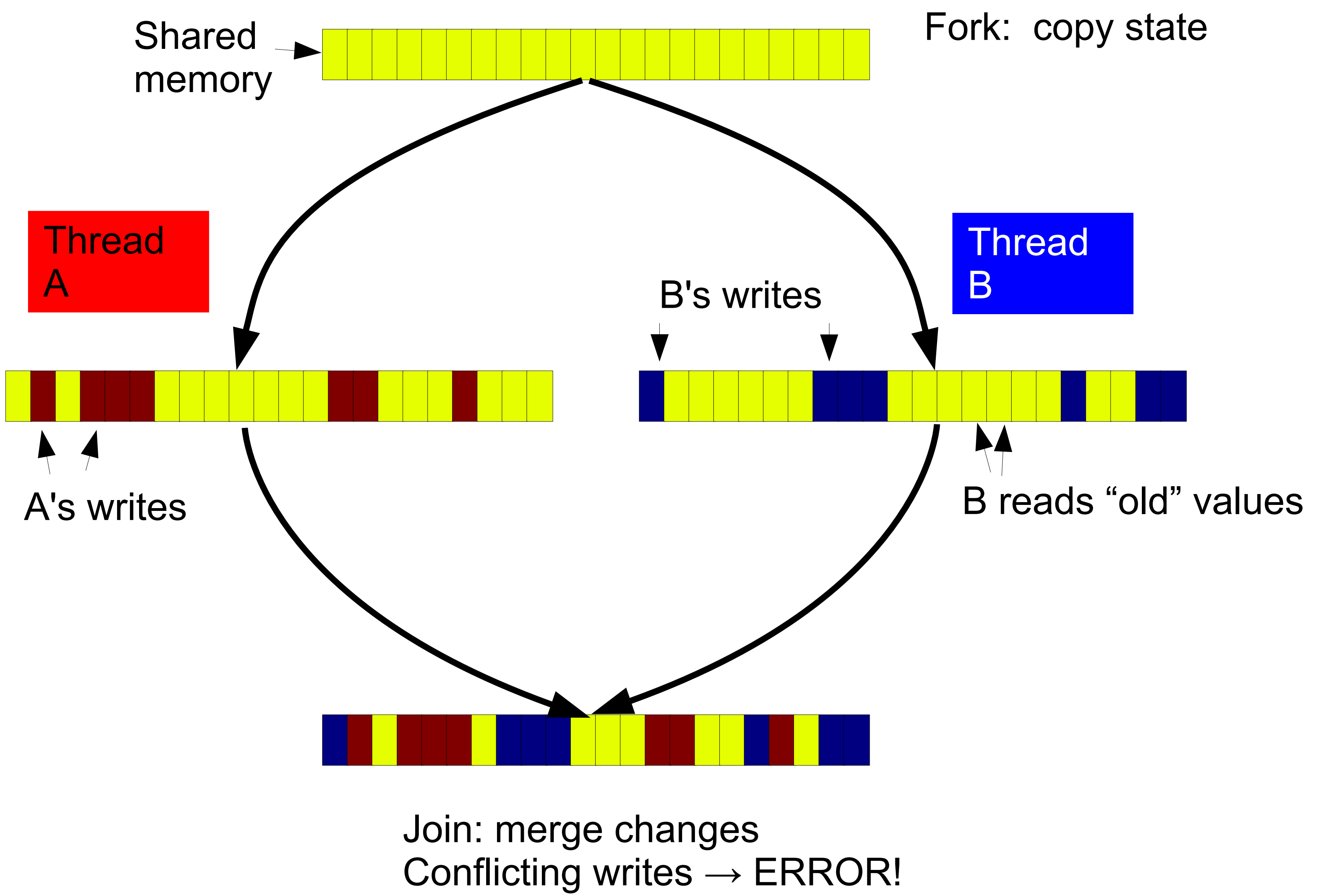
Nondeterministic Network
(For Contrast)

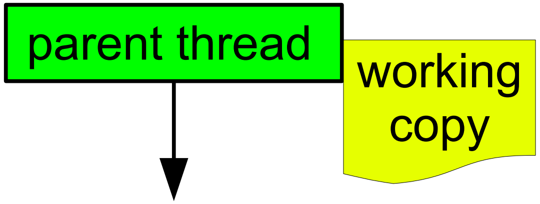
```
while(true) {
  result = receive(in);
  store(result);
  send(new_task(), out);
}
```

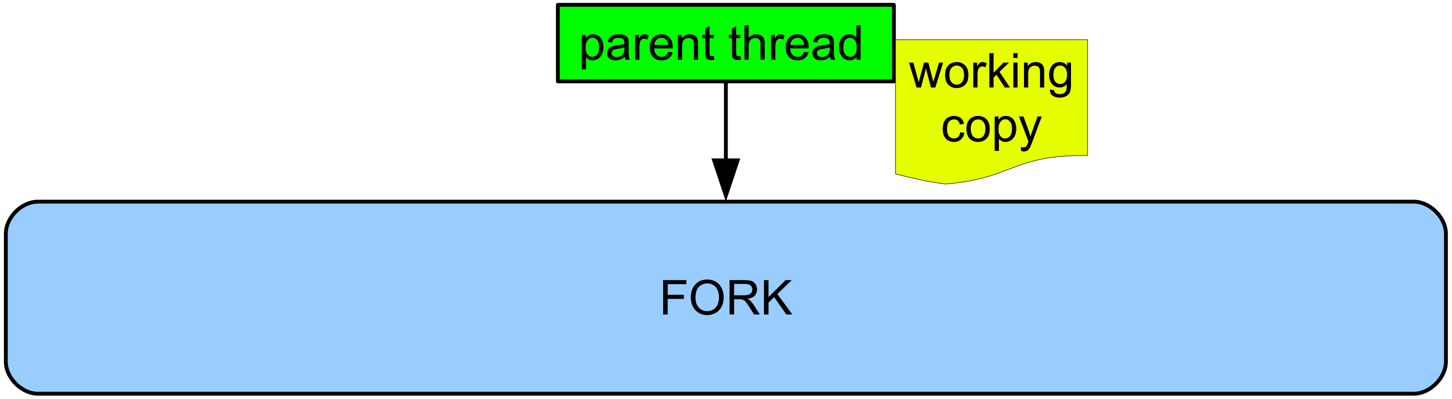


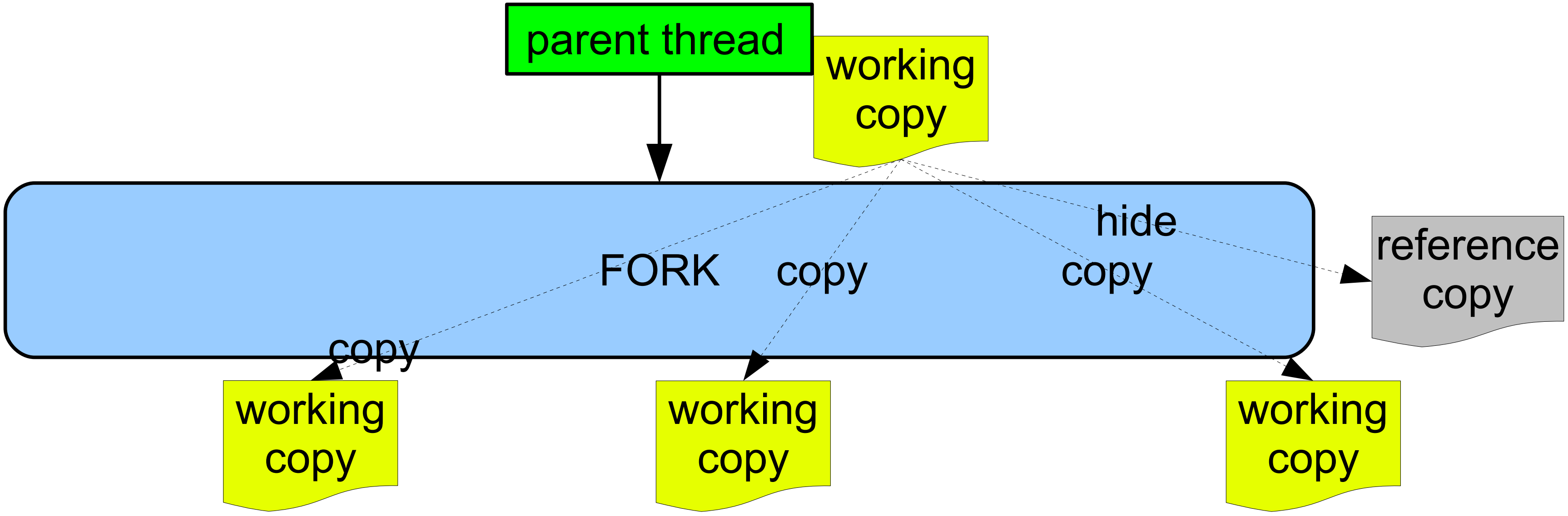

```
while(true) {
  task = receive(in);
  result = process(task);
  send(result, out);
}
```

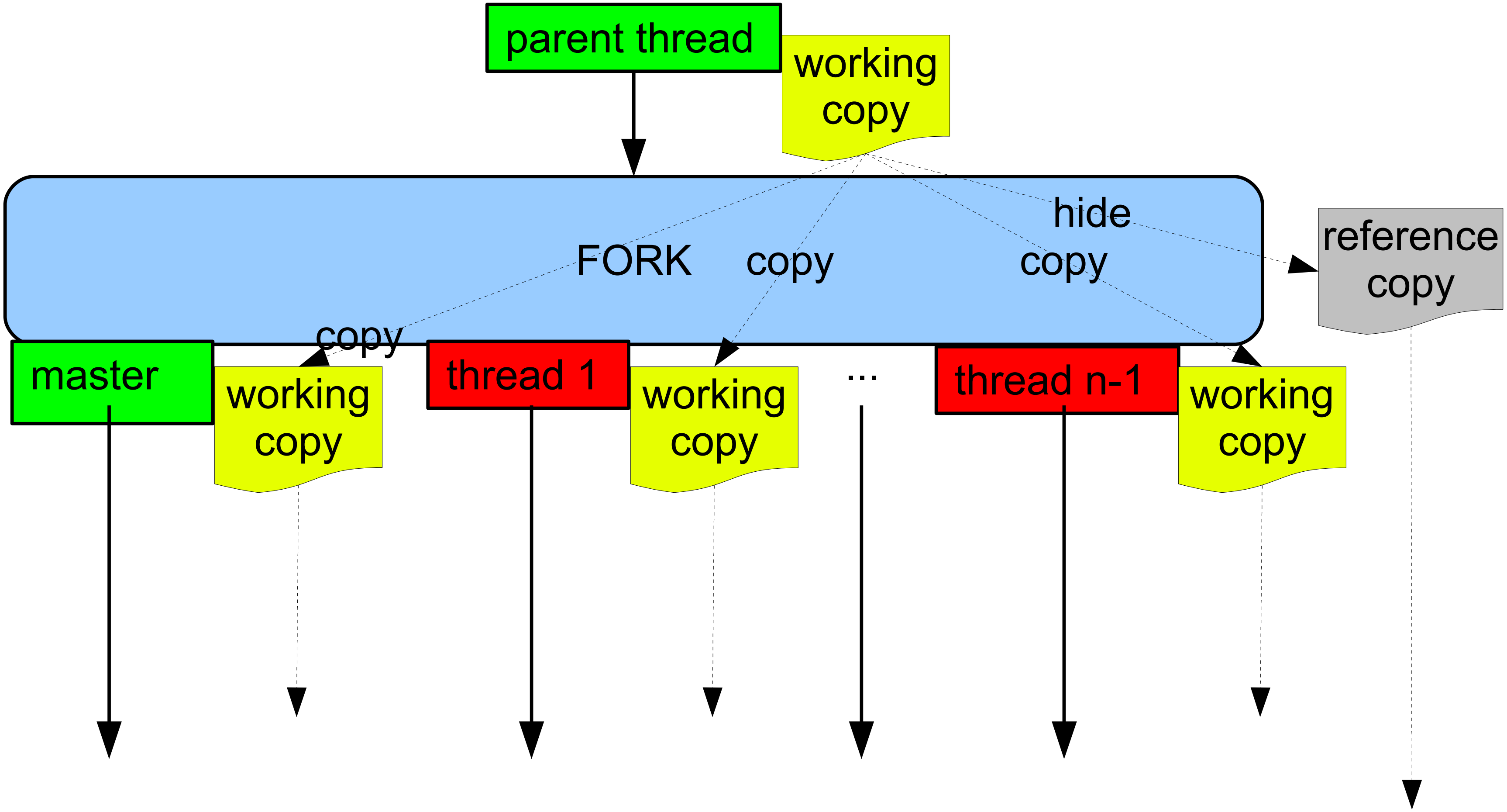
Working Copies Determinism

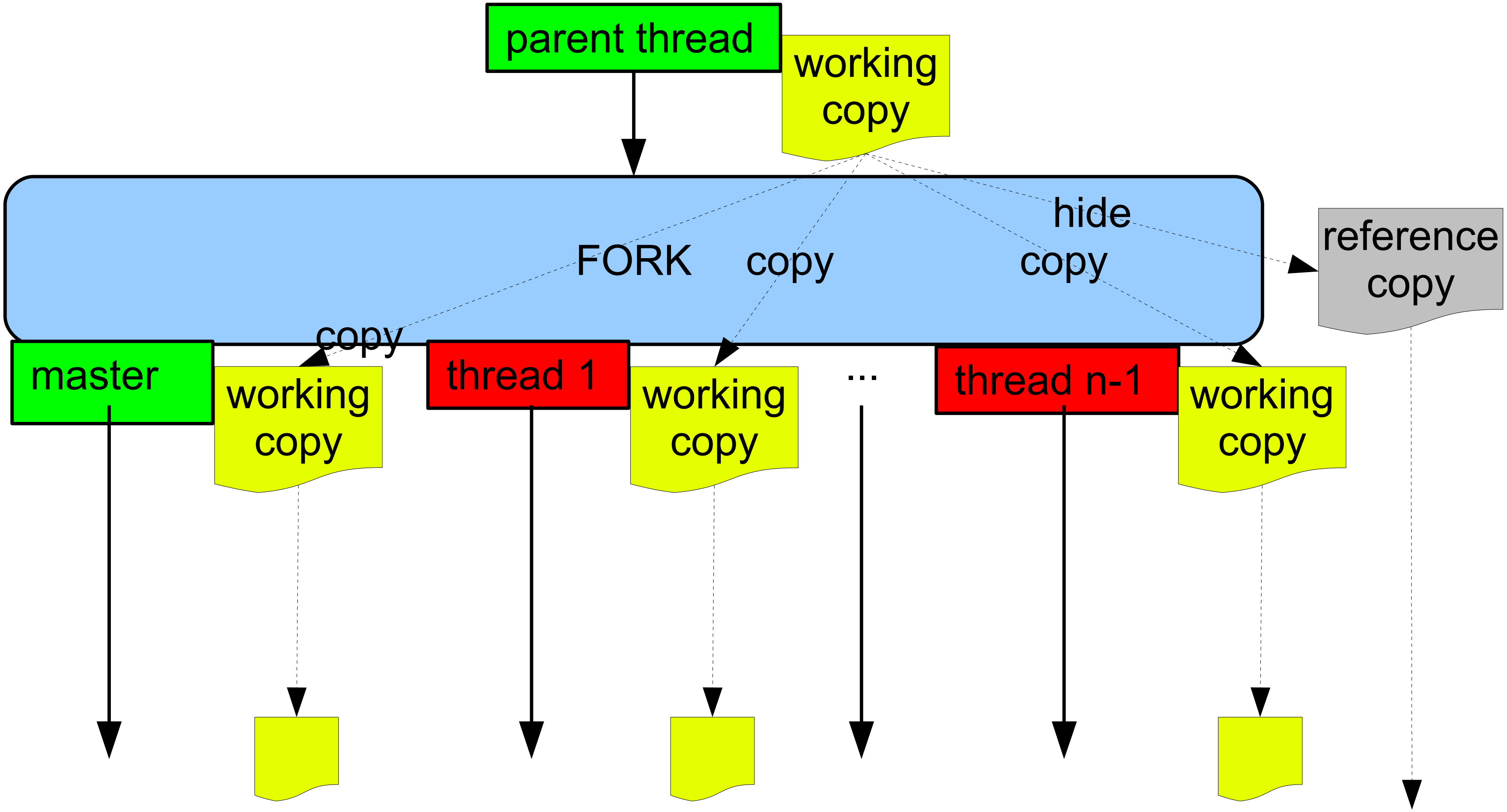


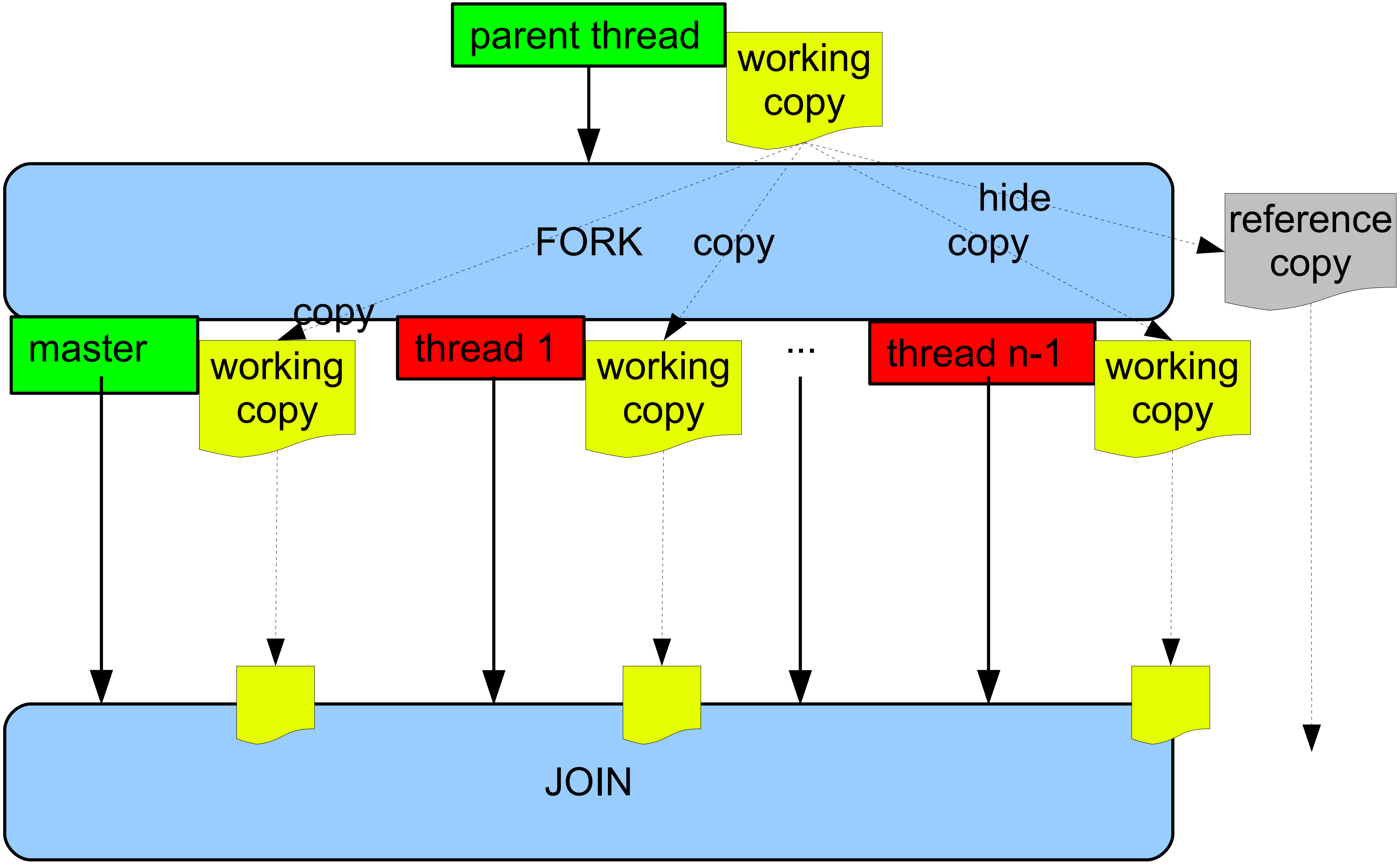

parent thread
working copy

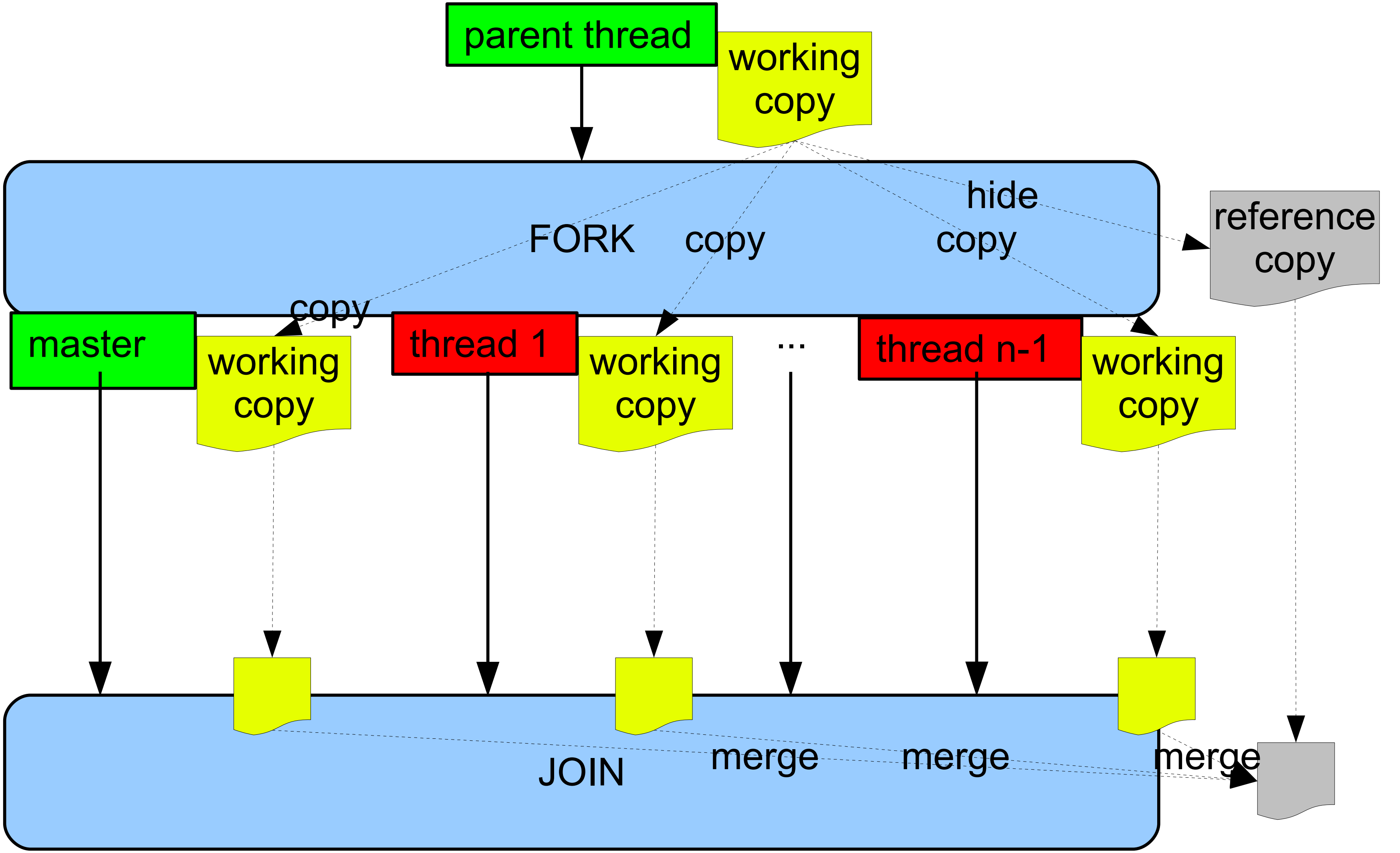

parent thread
working copy
FORK

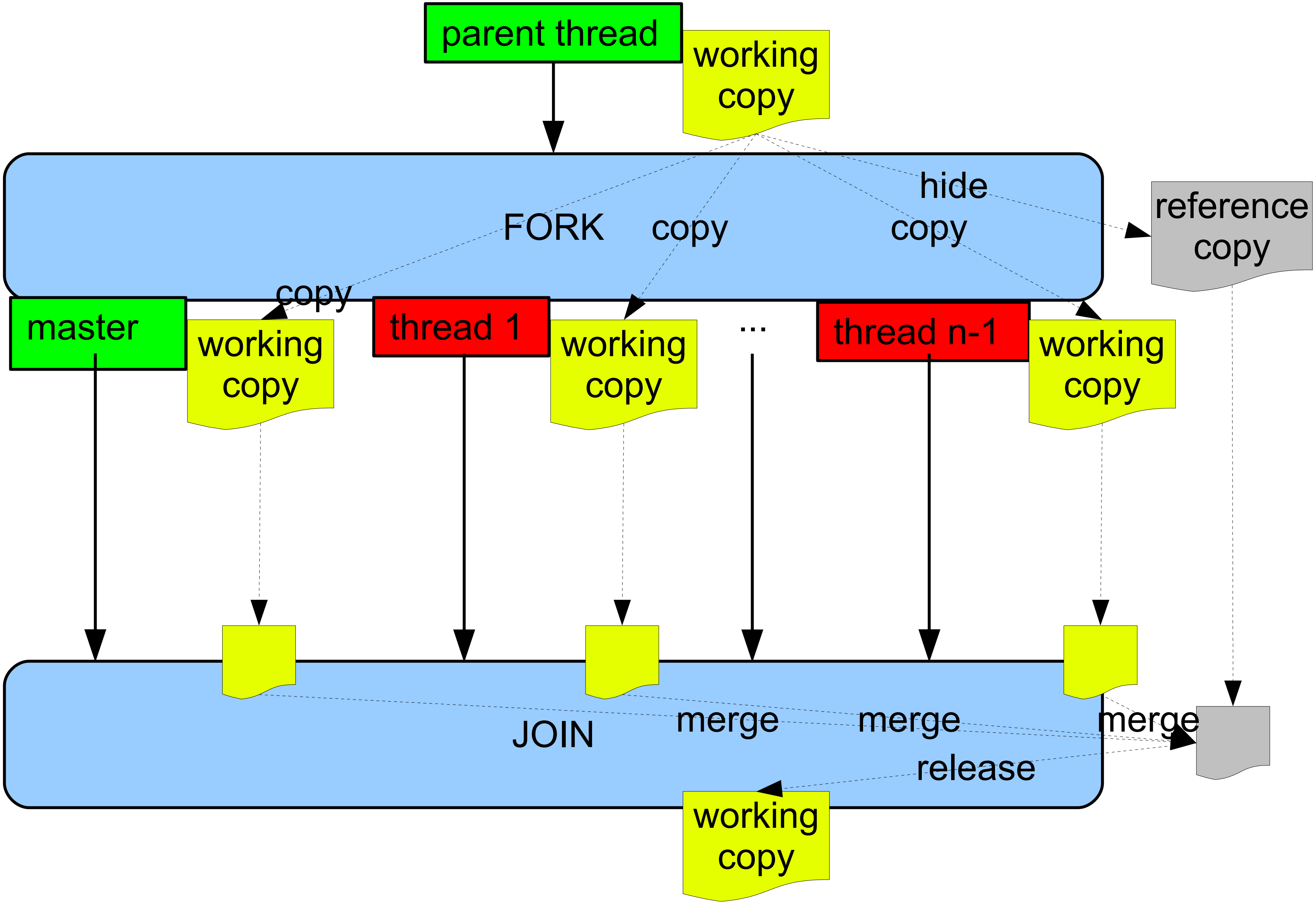

parent thread
working copy
FORK
copy
copy
hide
copy
reference copy
copy
working copy
working copy
working copy

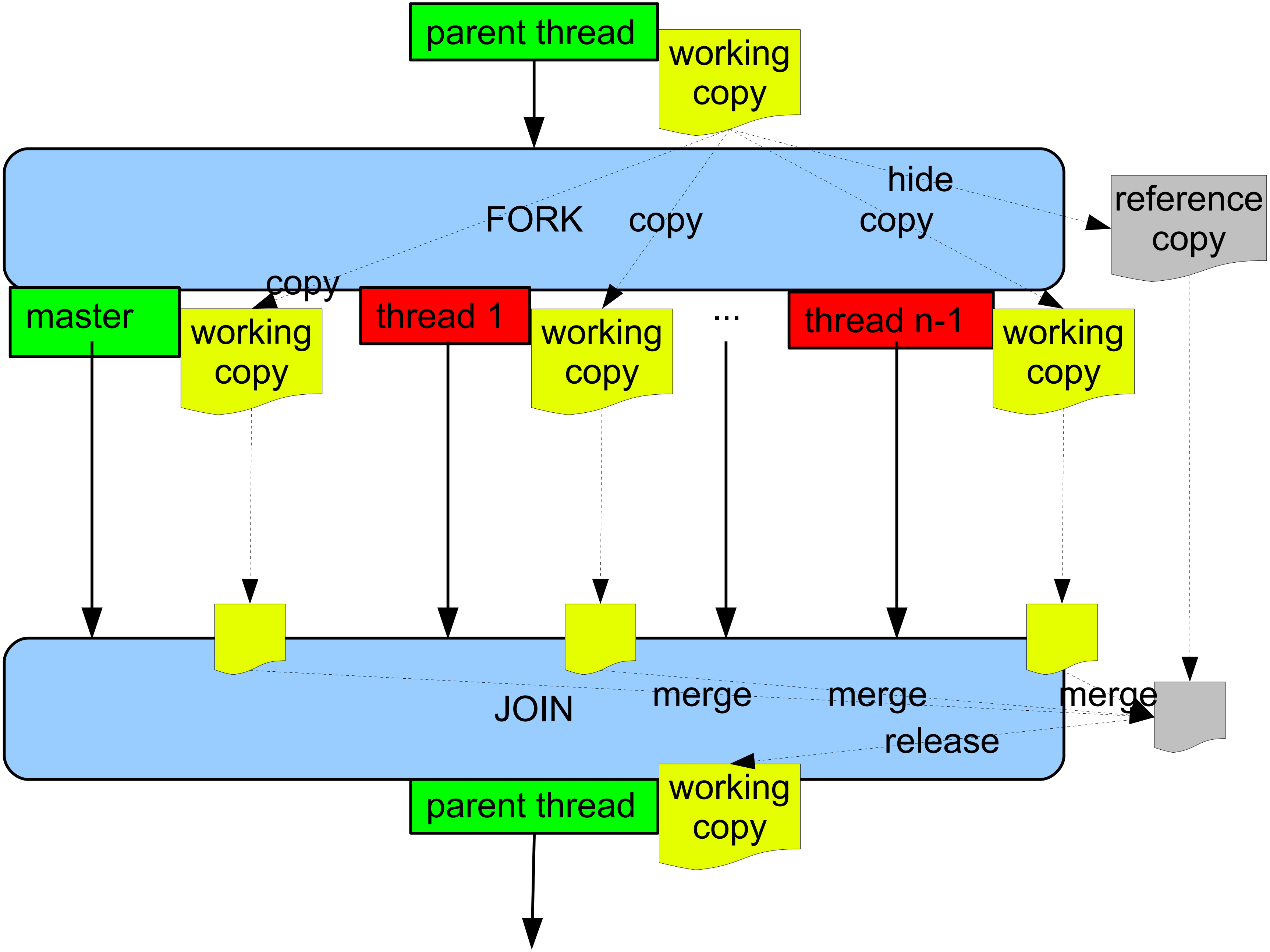

parent thread
working copy
hide
reference copy
FORK
copy
copy
copy
master
working copy
thread 1
working copy
...
thread n-1
working copy


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
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

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

SEQUENTIAL

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

Example

OpenMP

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

Creates new threads, distributes work

Joins threads to parent

Example

DOMP

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

Creates new threads, distributes work + copies of shared state

Merges copies of shared vars into parent's state and joins threads to parent

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):

```
      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*

```
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

```
#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

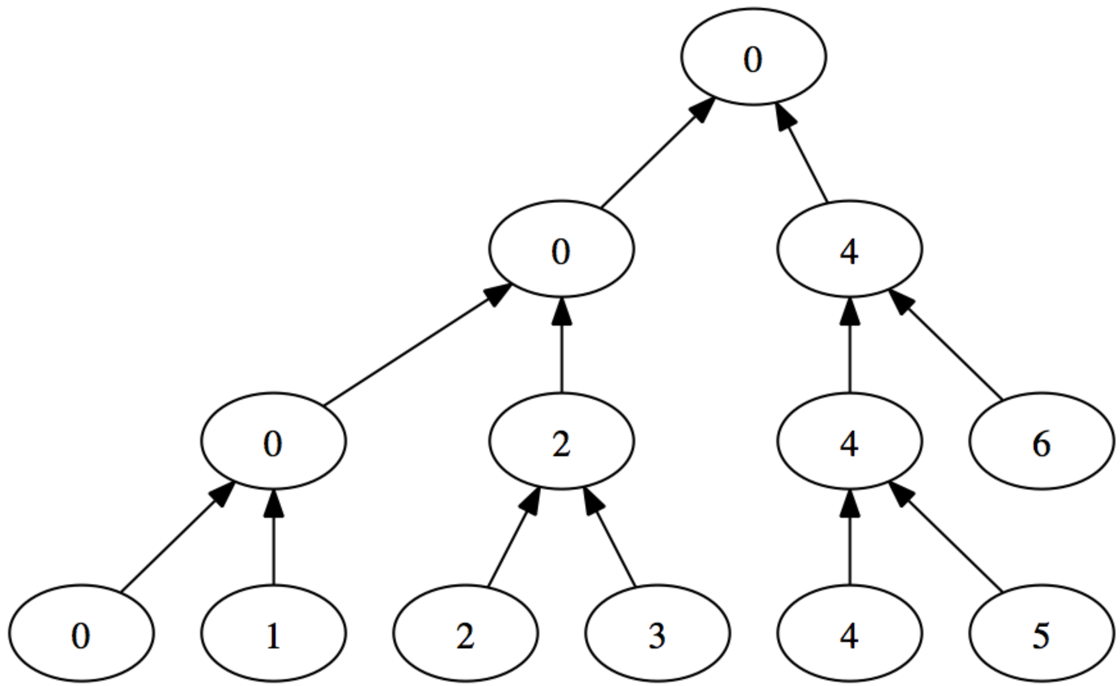
Naive Merge Loop

```
for each data segment seg in (stack, heap, bss)
  for each byte b in seg
    writer = WRITER_NONE
    for each thread t
      if (seg[t][b]] ≠ reference_copy[b])
        if (writer ≠ WRITER_NONE)
          race condition exception()
        writer = t
seg[MASTER][b] = seg[writer][b]
```

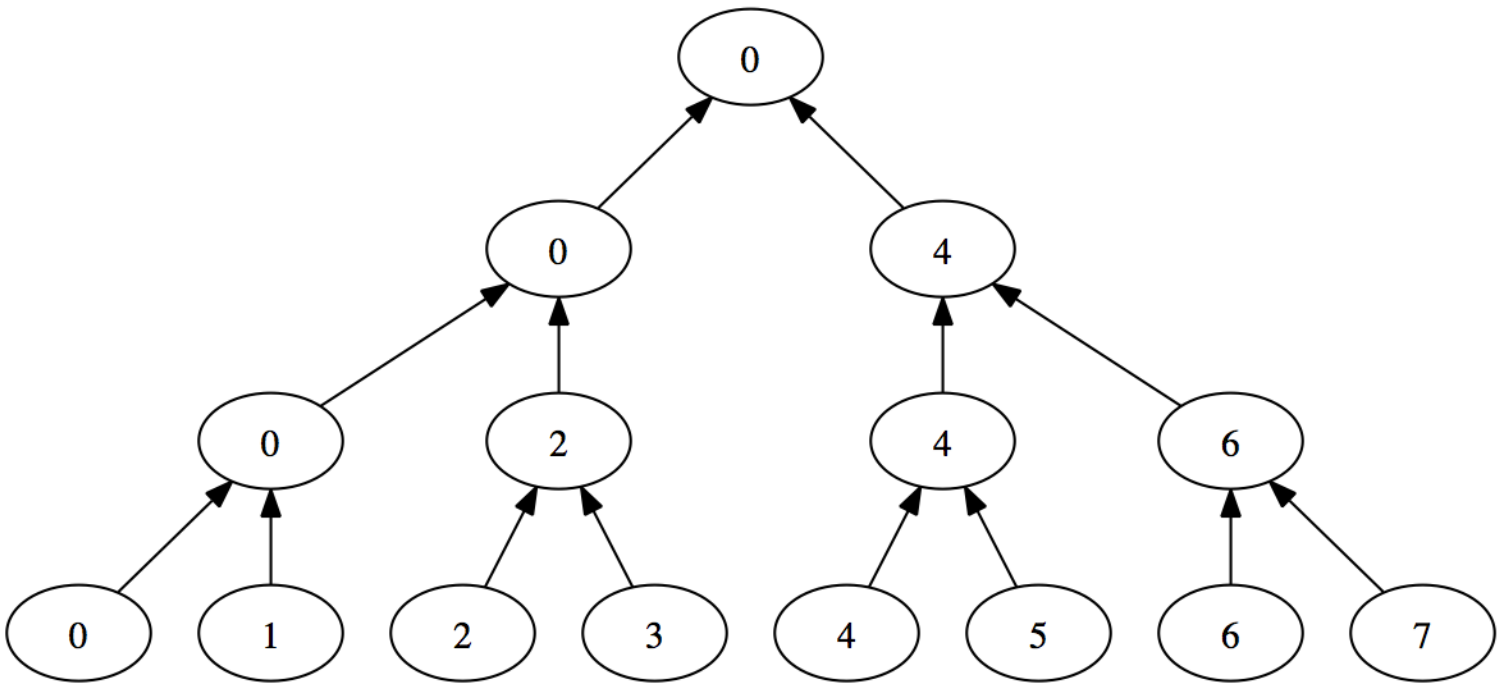
Improvements

- 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

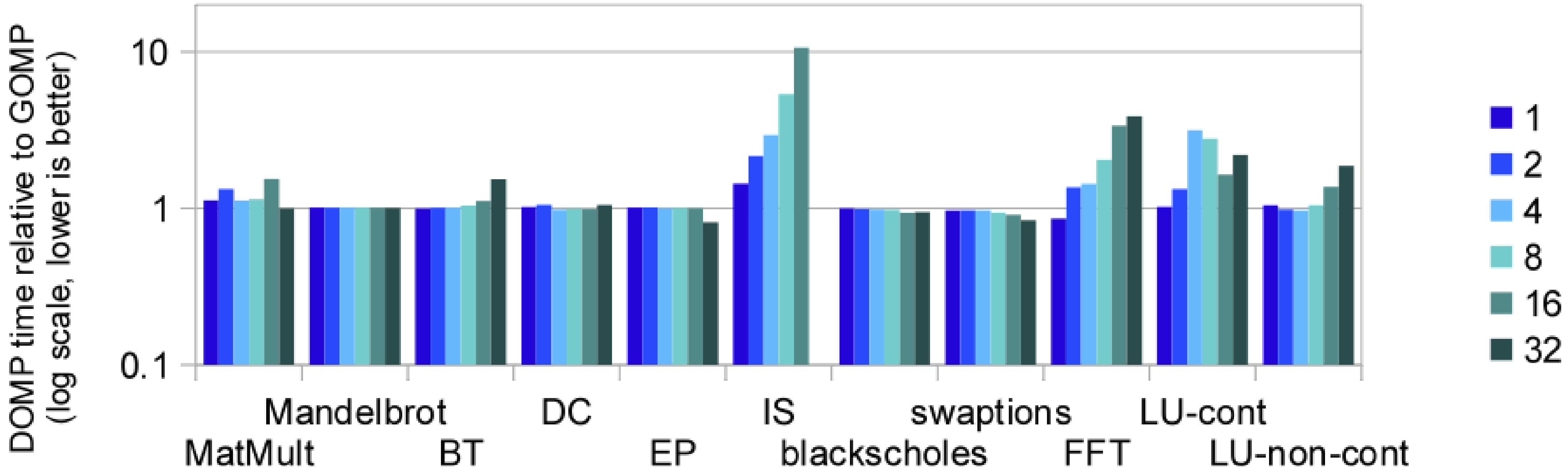
Outline

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

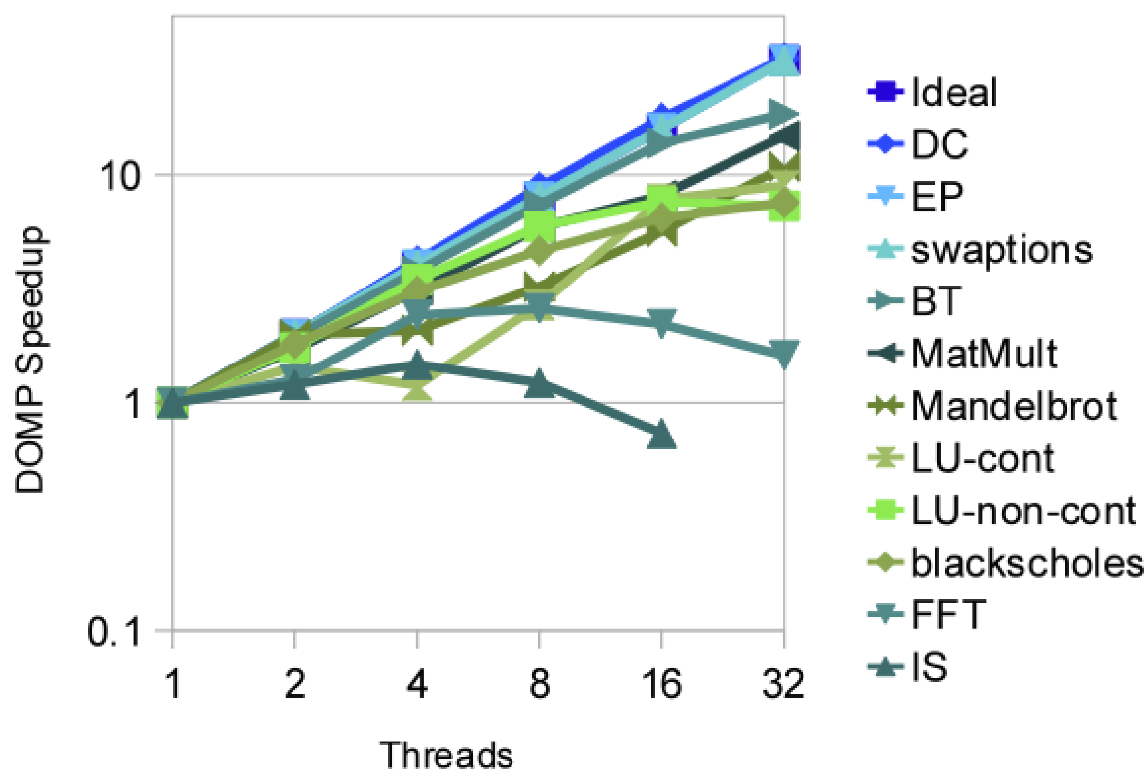
Outline

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

Performance



Speedup


DOMP Speedup
10
1
0.1
1
2
4
8
16
32
Threads
Ideal
DC
EP
swaptions
BT
MatMult
Mandelbrot
LU-cont
LU-non-cont
blackscholes
FFT
IS

Why Is IS So Bad?

Benchmark	Max Pages	Total Pages
MatMult	24578	24578
Mandelbrot	1	1
BT	4	1911
DC	2	3
EP	3	4
IS	34778	90100
blackscholes	9768	9768
swaptions	677	677
FFT	5	5
LU-cont	7	7
LU-non-cont	7	7

Converting Nondeterministic Code

	Total	DOMP Changes	Module	%
MatMult	109	0	0	0
Mandelbrot	105	0	0	0
BT	3589	16	30	1
DC	2809	3	48	2
EP	228	16	30	20
IS	634	0	0	0
blackscholes	359	0	0	0
swaptions	1780	0	0	0
FFT	1504	0	0	0
LU-cont	2484	0	0	0
LU-non-cont	1890	0	0	0

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.