CoreDet:
A Compiler and Runtime System for Deterministic Multithreaded Execution

Tom Bergan

Owen Anderson, Joe Devietti, Luis Ceze, Dan Grossman

A Multithreaded Program

```
global x=0
```

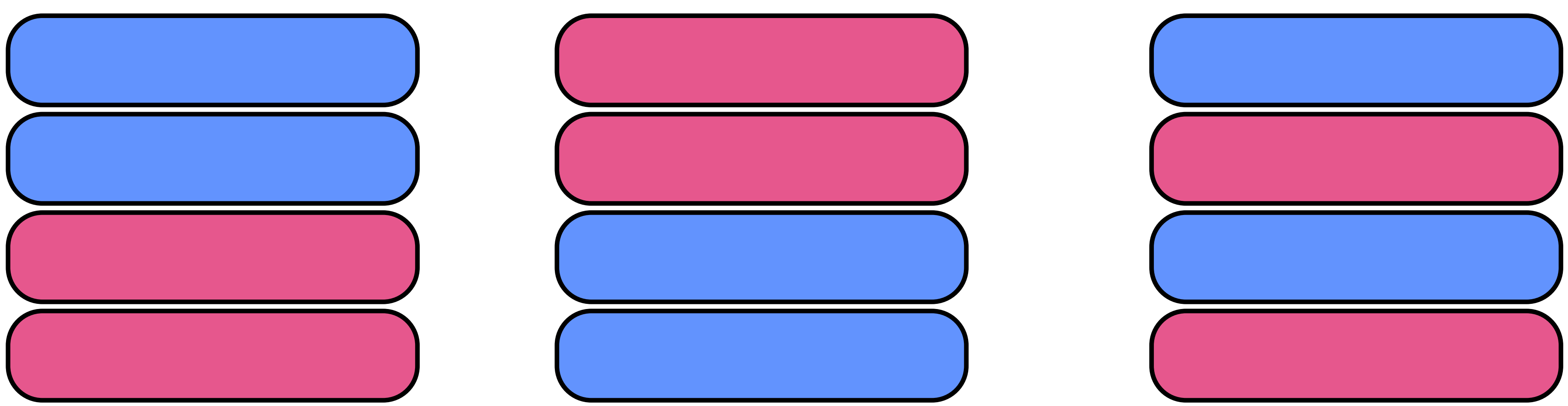
Thread 1

```
t := x
x := t + 1
```

Thread 2

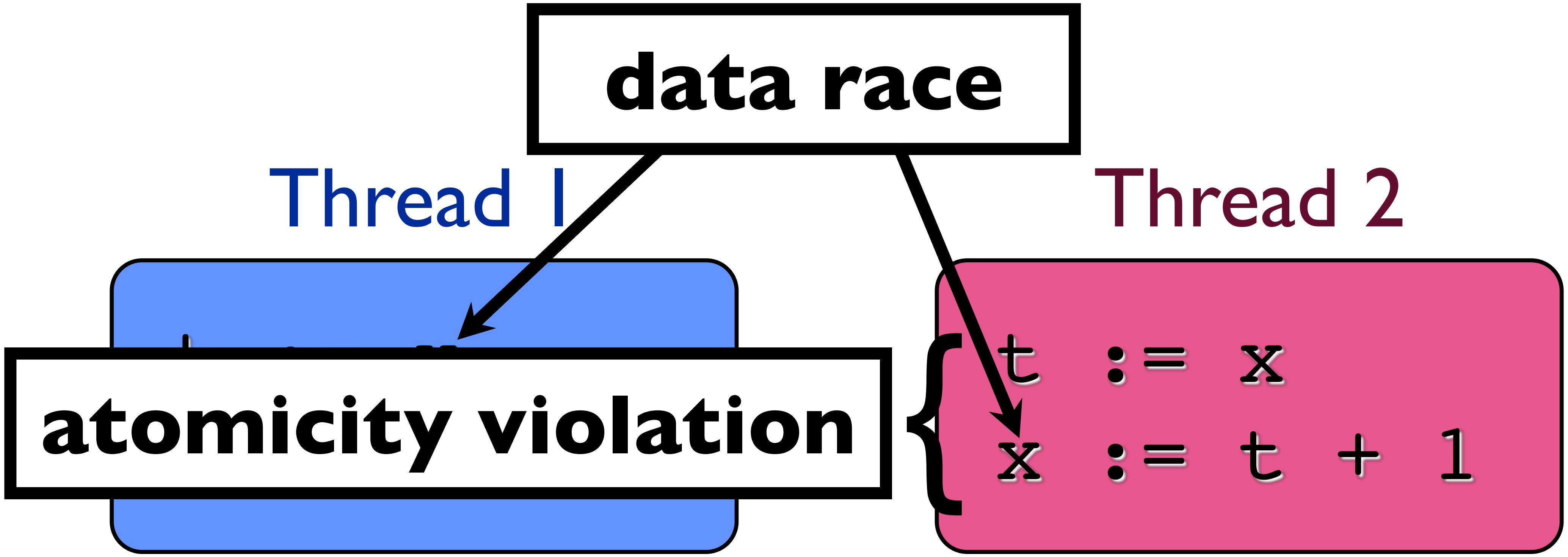
```
t := x
x := t + 1
```

What is x?



`x == 2` `x == 2` **`x == 1`**

A Multithreaded Program



What is x?

We're not trying to make these bugs go away

We're trying to make them come back!

x == 2 x == 2 x == 1

Another Multithreaded Program

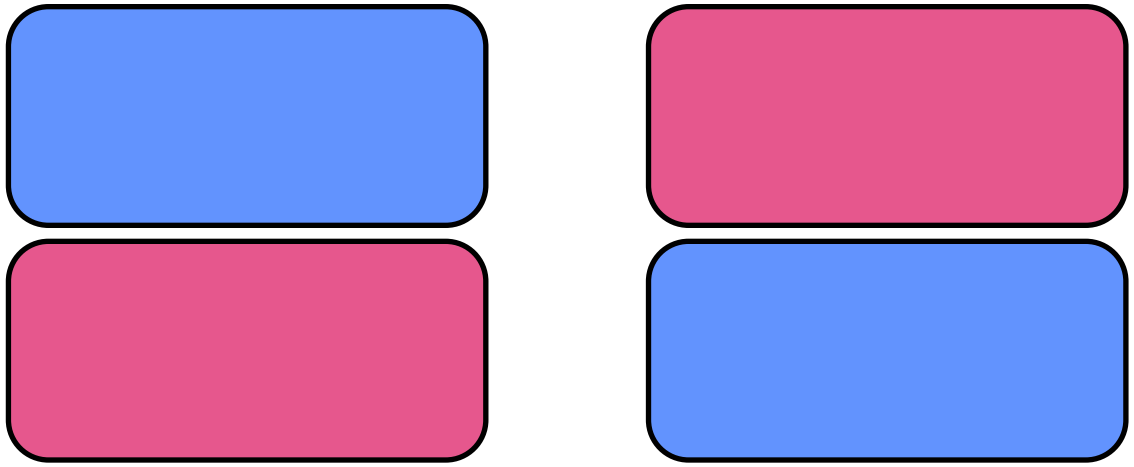
```
global x=0
```

Thread 1

```
lock(L)
assert(x!=42)
unlock(L)
```

Thread 2

```
lock(L)
x := 42
unlock(L)
```



no bug BUG!

The Problem With Multithreading

- Shared-memory access interleavings are a hidden source of nondeterminism

hard to test

hard to debug

hard to replicate

Determinism Can Help!

hard to test

- ✓ test inputs, not interleavings
- ✓ software behaves as tested

hard to debug

- ✓ no more heisenbugs!
- ✓ reproduce bugs from the field

hard to replicate

- ✓ easy to synchronize replicas

Deterministic MultiProcessing

Goal: deterministic execution ...

- of arbitrary multithreaded programs
- without sacrificing scalability

Eliminate shared-memory nondeterminism

- execution is a function of inputs (including I/O)

DMP [prior work, ASPLOS'09]:

- **hardware architecture** for determinism
- using ownership-tracking and transactions

CoreDet

CoreDet: deterministic execution ...

- of arbitrary, unmodified C/C++ pthreads programs
- **without special hardware**
- without sacrificing scalability

CoreDet

CoreDet: deterministic execution ...

- of arbitrary, unmodified C/C++ pthreads programs
- without special hardware
- **without sacrificing scalability**

CoreDet

CoreDet: deterministic execution ...

- of arbitrary, unmodified C/C++ pthreads programs
- without special hardware
- without sacrificing scalability

Contributions:

- new algorithm for deterministic execution
  - uses *store-buffering* and *relaxed memory consistency*
- compiler (LLVM pass) and a runtime library
  - static optimizations
  - dealing with external code

Related Work

helps with ...	Record + Replay	Kendo	DMP	**CoreDet**
... testing?	○			
... debugging?	●			
... replication?	○ / ●			
assumes race free?	sometimes			
needs hw?	usually	no	yes	**no**
examples:	FDR, Rerun, Respec	[ASPLOS'09]	[ASPLOS'09]	

FDR, Rerun [ISCA'03,'08]:

- offline replay (for debugging)
- in hardware

Respec [ASPLOS'10]:

- online replay (for replicas)
- in software

sync-only determinism

Outline

Recap of DMP [ASPLOS'09]:

- DMP-Ownership
- DMP-TM
- What's wrong with doing these in software?

CoreDet:

less complexity than DMP-TM
with comparable scalability
not sequentially consistent!

- DMP-Buffering
- Performance Evaluation

DMP-Serial [ASPLOS'09]

quantum

Thread 1

```
a := x
b := y
```

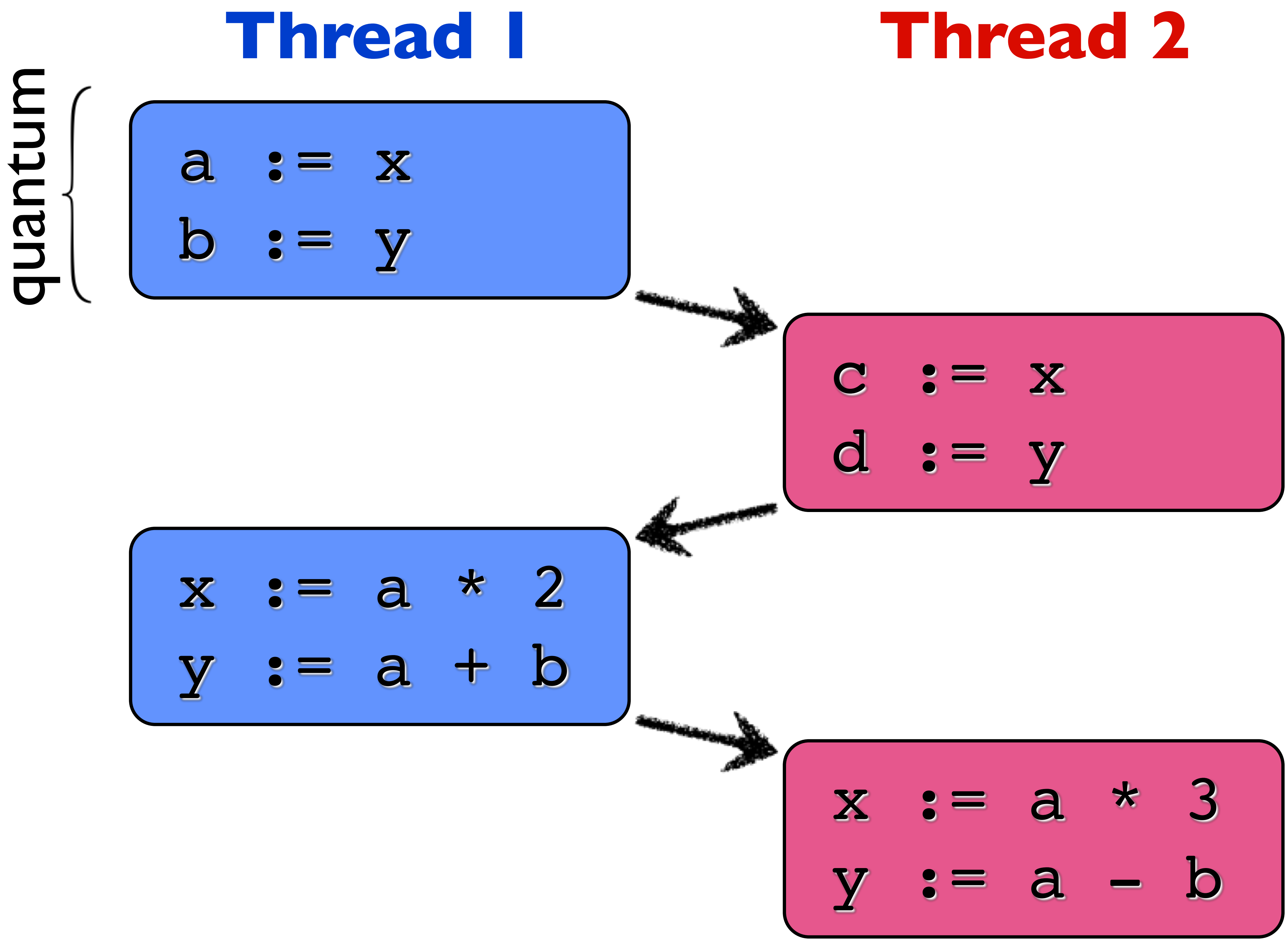
```
x := a * 2
y := a + b
```

Thread 2

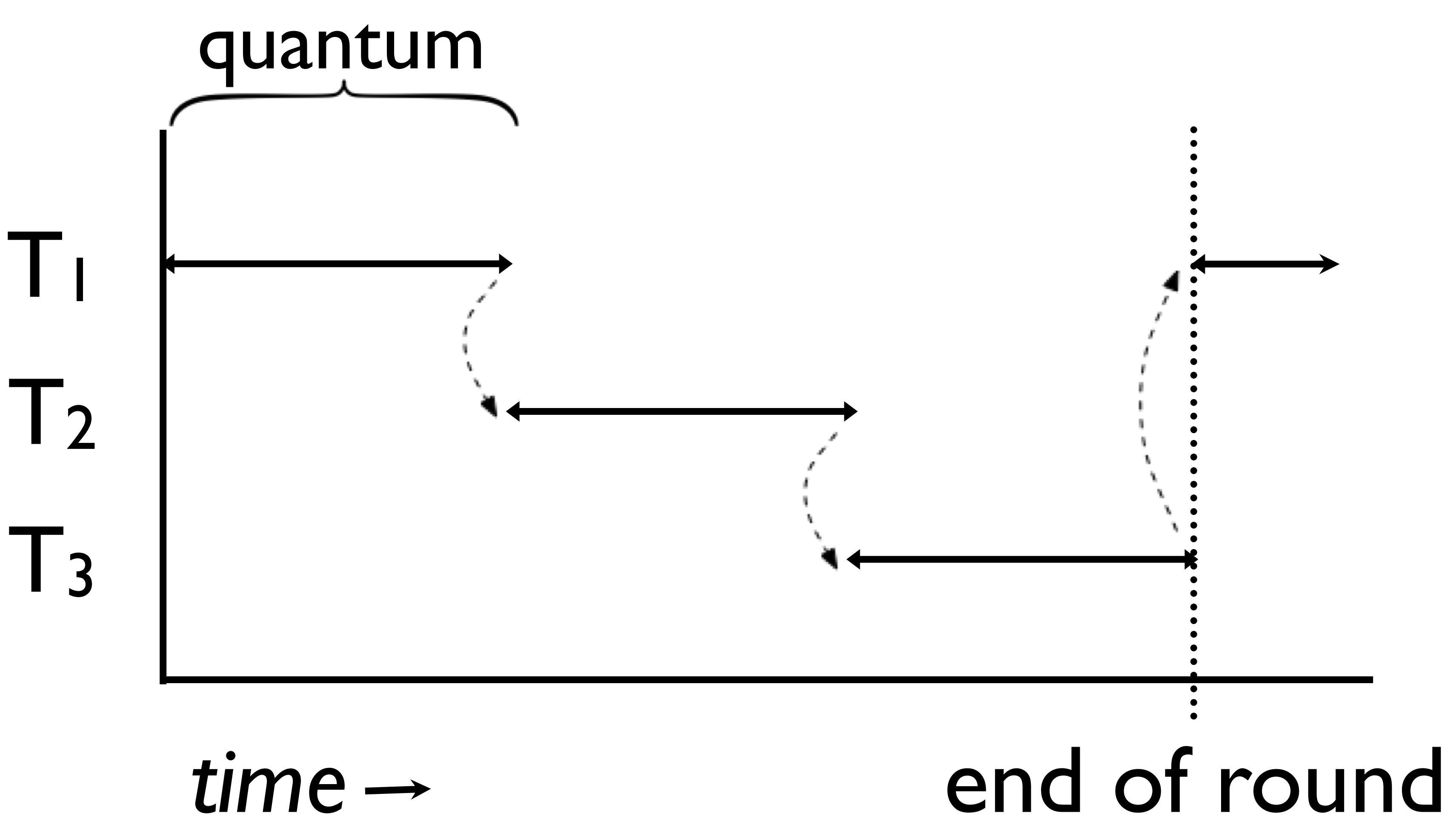
```
c := x
d := y
```

```
x := a * 3
y := a - b
```

DMP-Serial [ASPLOS'09]

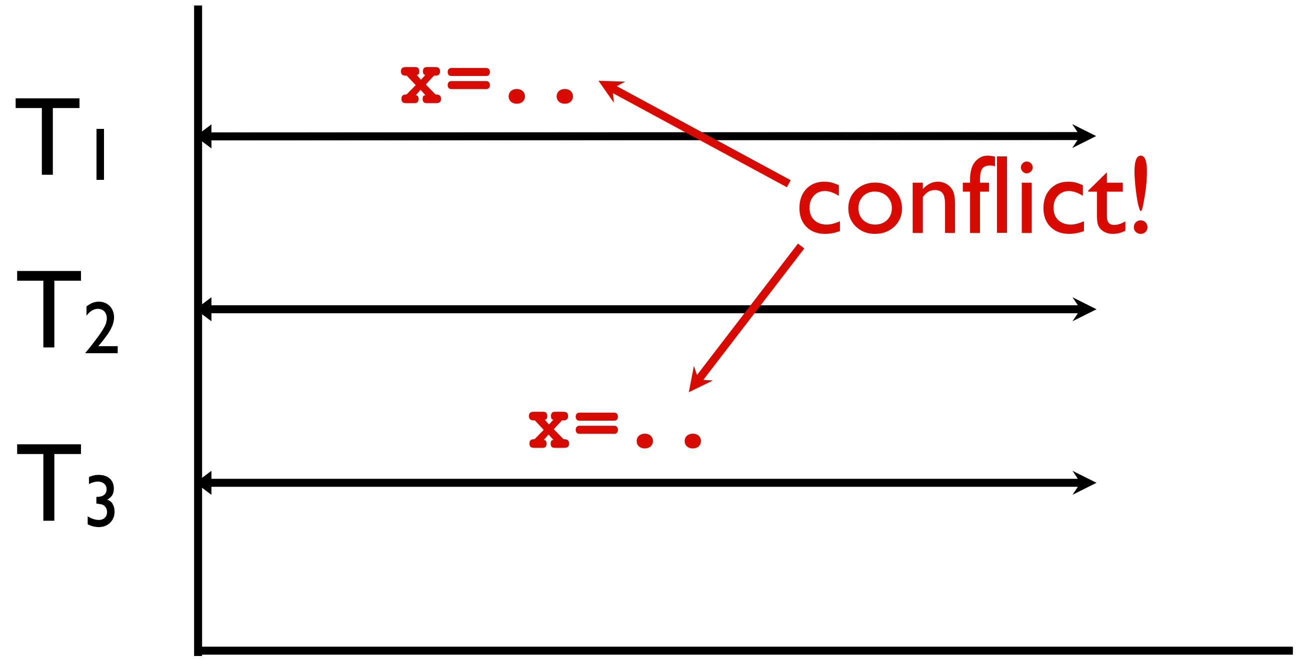


DMP-Serial [ASPLOS'09]



Execution is completely serialized

Recovering Parallelism [ASPLOS'09]



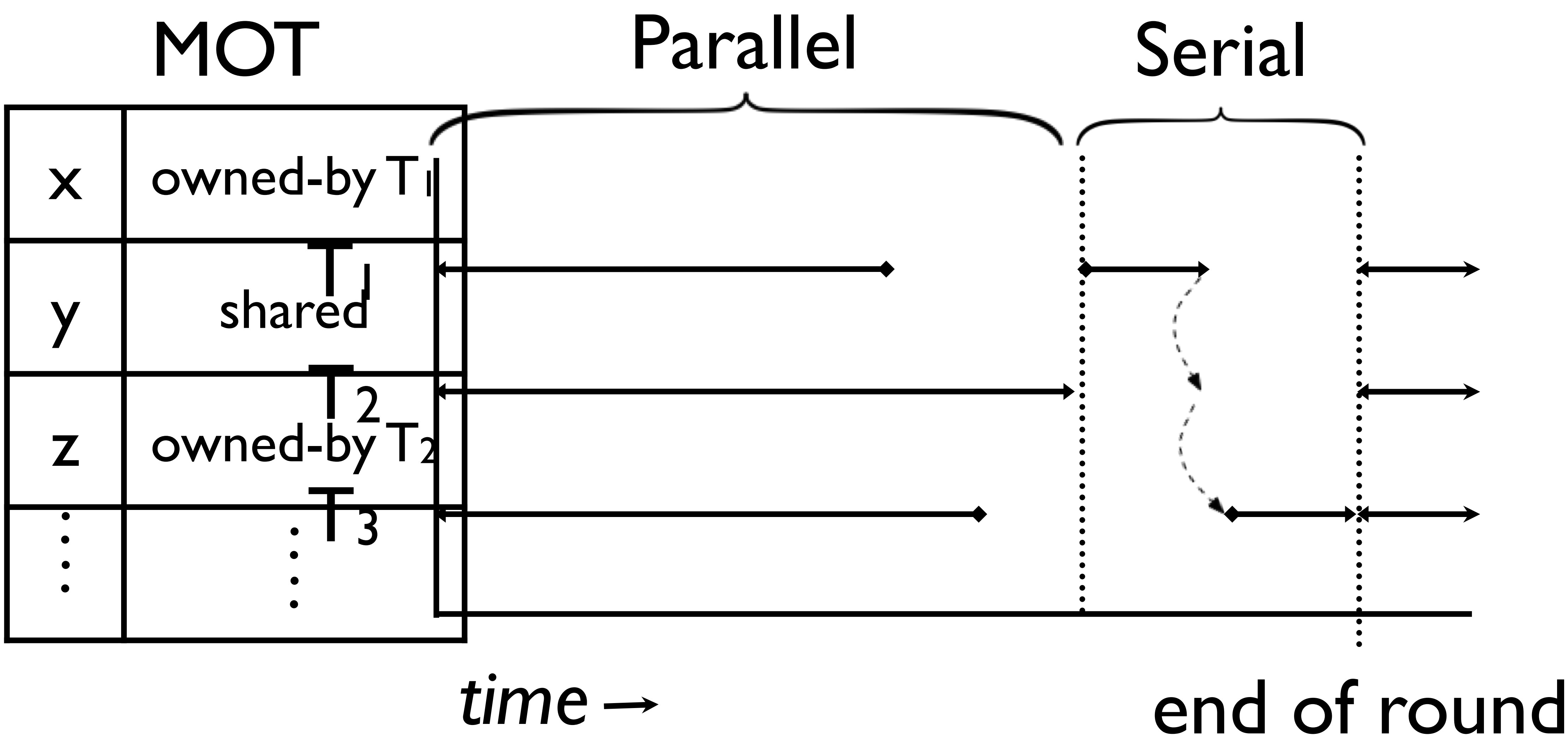
To recover parallelism …

… must resolve conflicts deterministically

by partitioning ownership (DMP-Ownership)

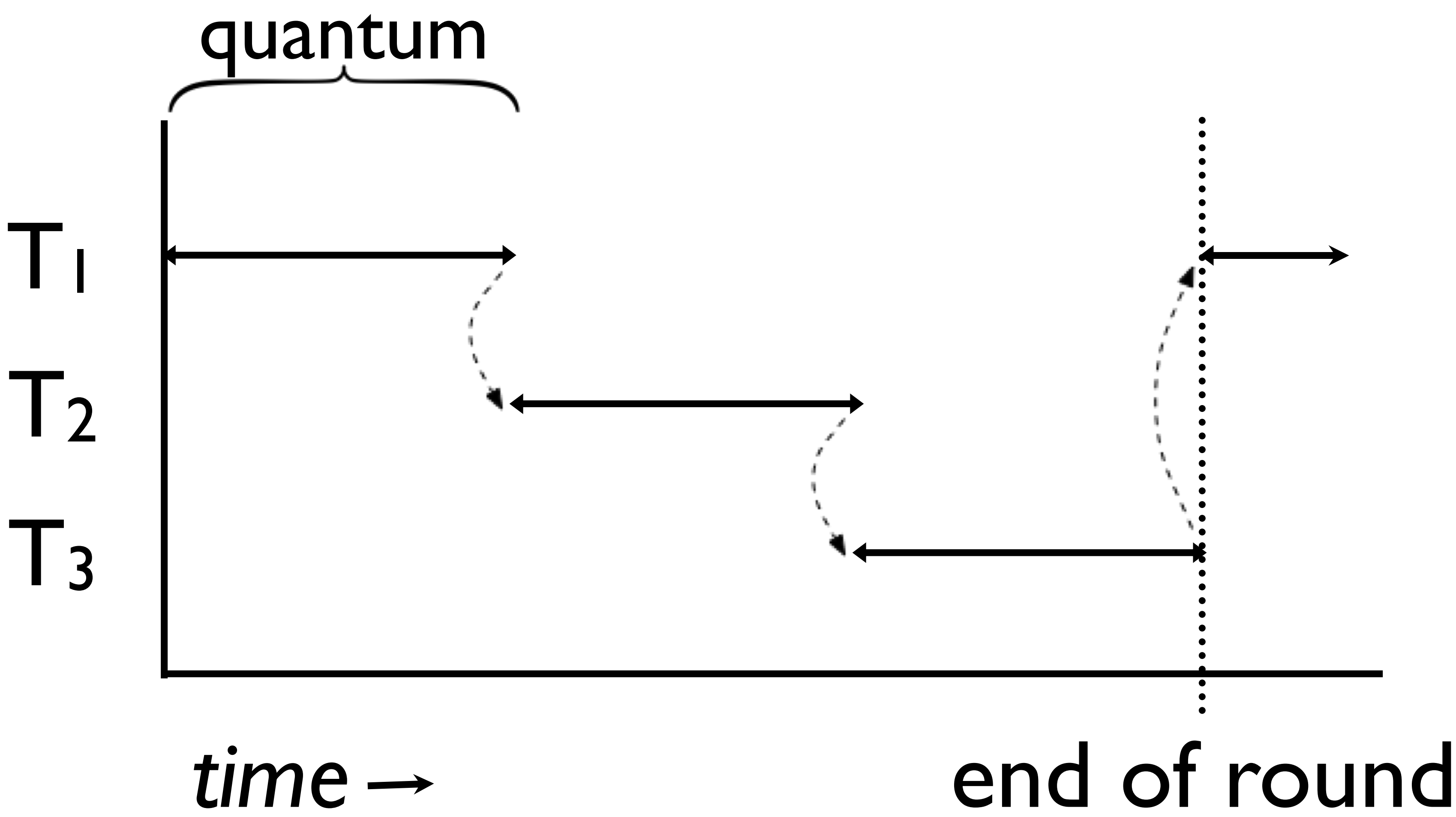
by using transactions (DMP-TM)

DMP-Ownership [ASPLOS'09]



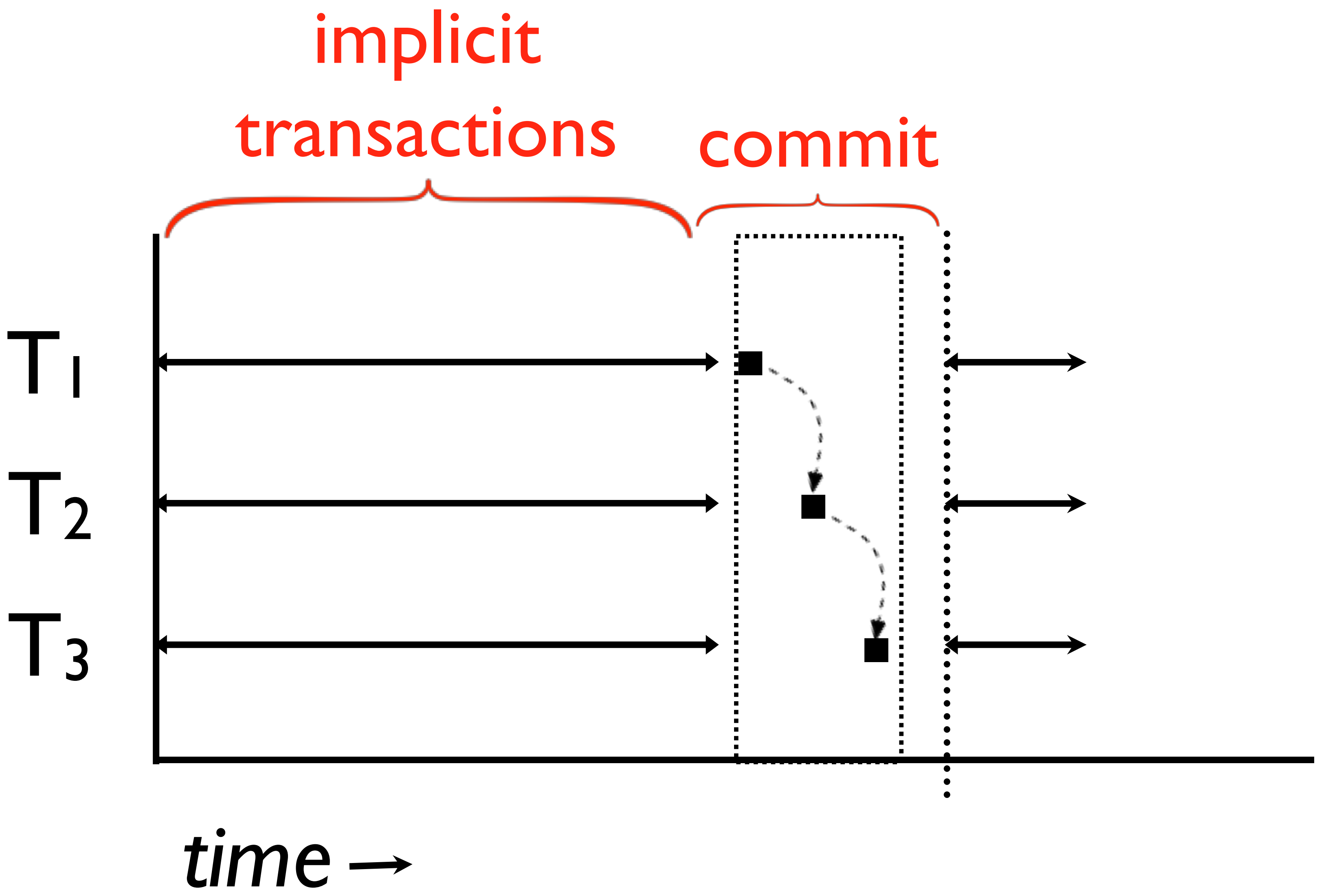
Parallel mode: no communication (can write only to private data)
Serial mode: arbitrary communication

DMP-TM [ASPLOS'09]



Start with DMP-Serial, then add transactions . . .

DMP-TM [ASPLOS'09]



Execution is parallel and transactional

DMP in software

Can we implement DMP-Ownership in CoreDet?

- ✓ yes (we have!)
- X sub-optimal scalability
  (too conservative about what can run in parallel)

Can we implement DMP-TM in CoreDet?

- X not efficiently
  why not use STM?

DMP-TM in software

What's wrong with STM?

DMP-TM breaks important STM assumptions, specifically . . .

1) Transactions are rare
2) Transactions are short
3) Transactions are scoped

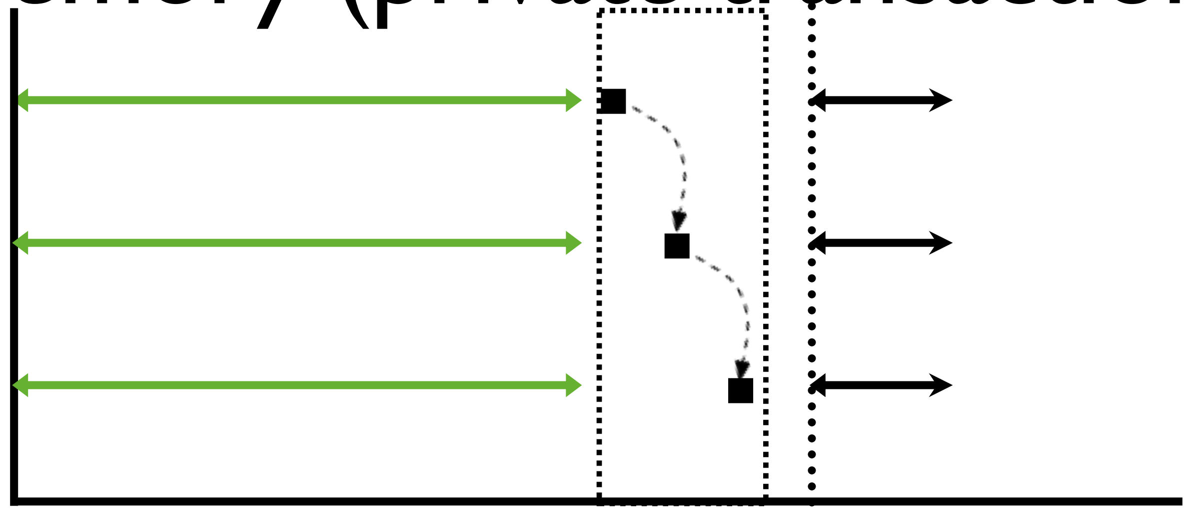
An unscoped transaction:

```
void foo() {
  ...
  begin_transaction()
  return
}
```

DMP-TM: what can we learn?

Speculation makes things hard

Good scalability by allowing parallel updates of versioned memory (private transaction buffers)



CoreDet's Insight:

Enable parallel updates without requiring speculation

Outline

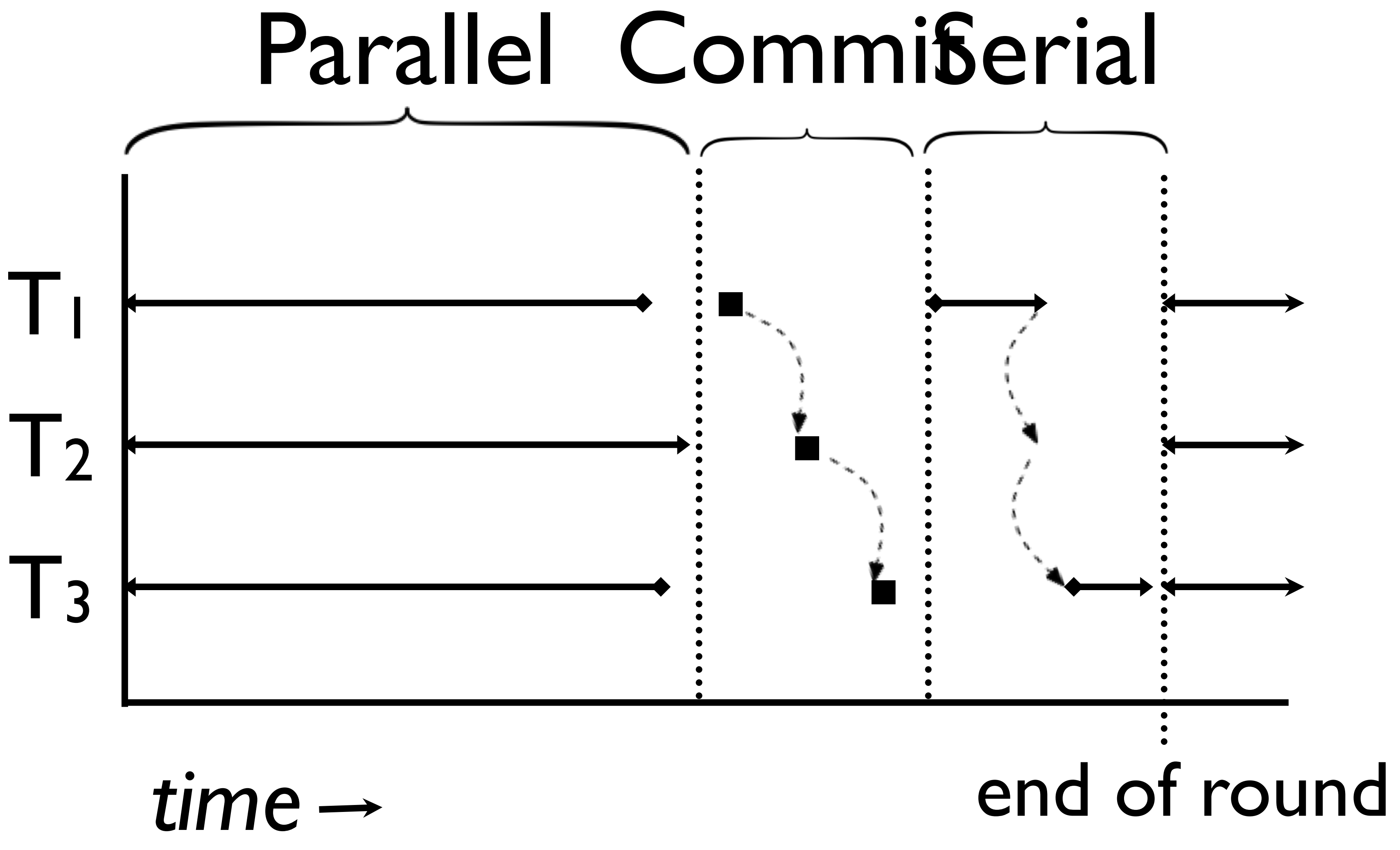
Recap of DMP [ASPLOS'09]:

- DMP-Ownership
- DMP-TM
- What's wrong with DMP in software?

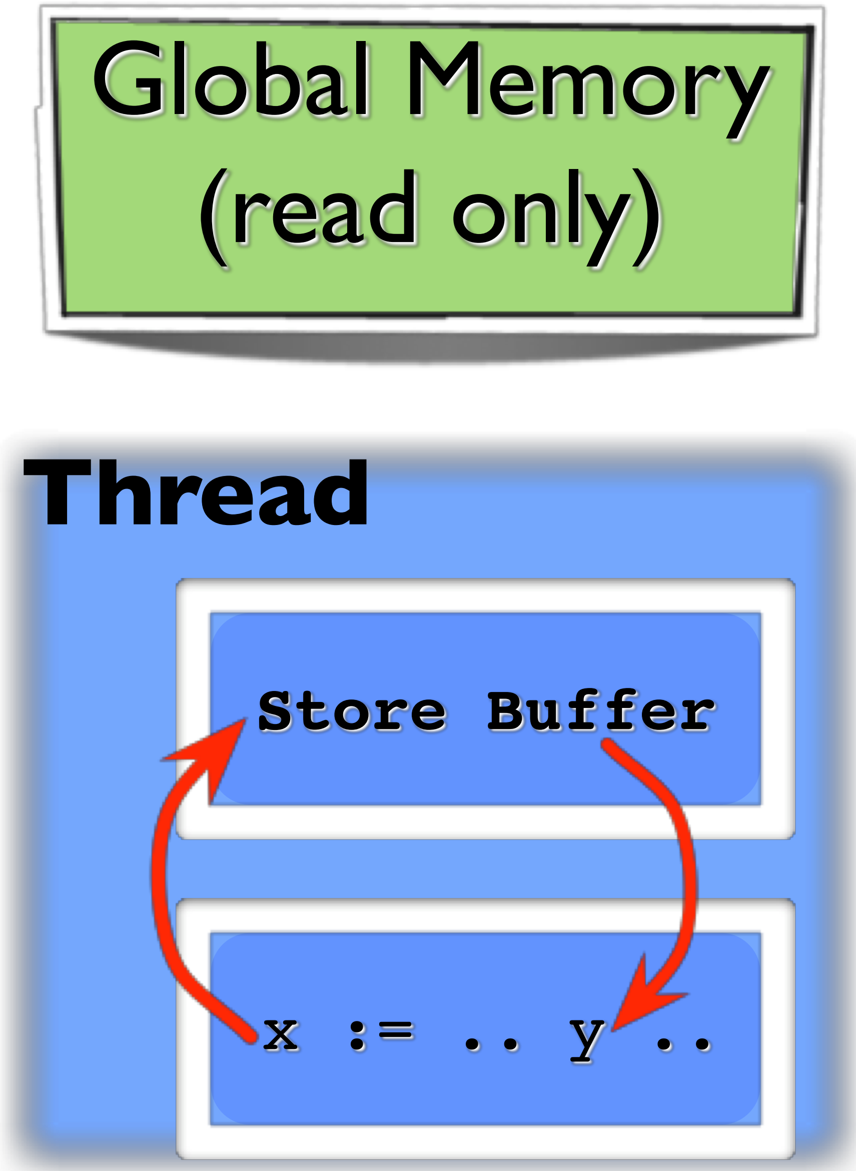
CoreDet:

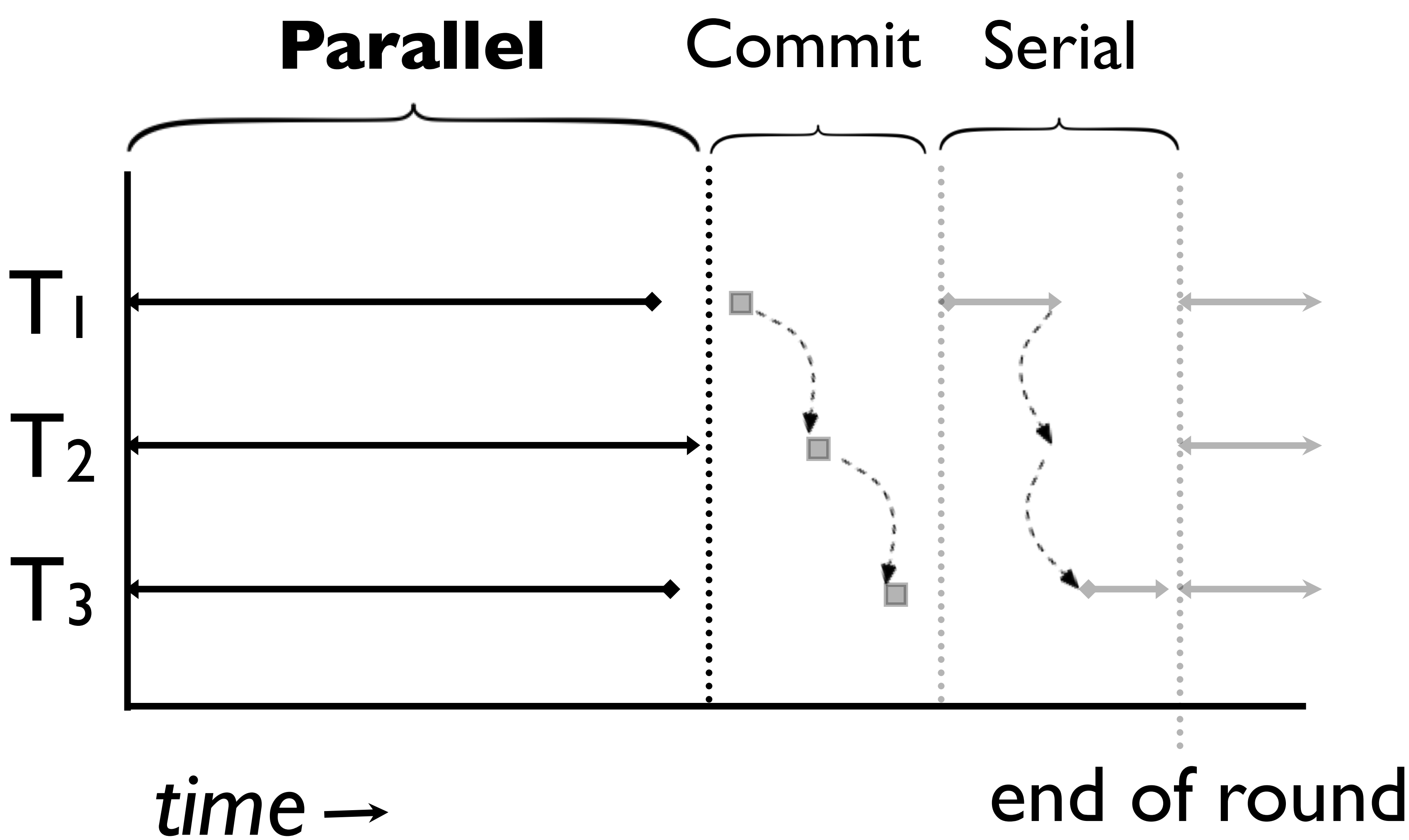
- **DMP-Buffering**
- Performance Evaluation

DMP-Buffering



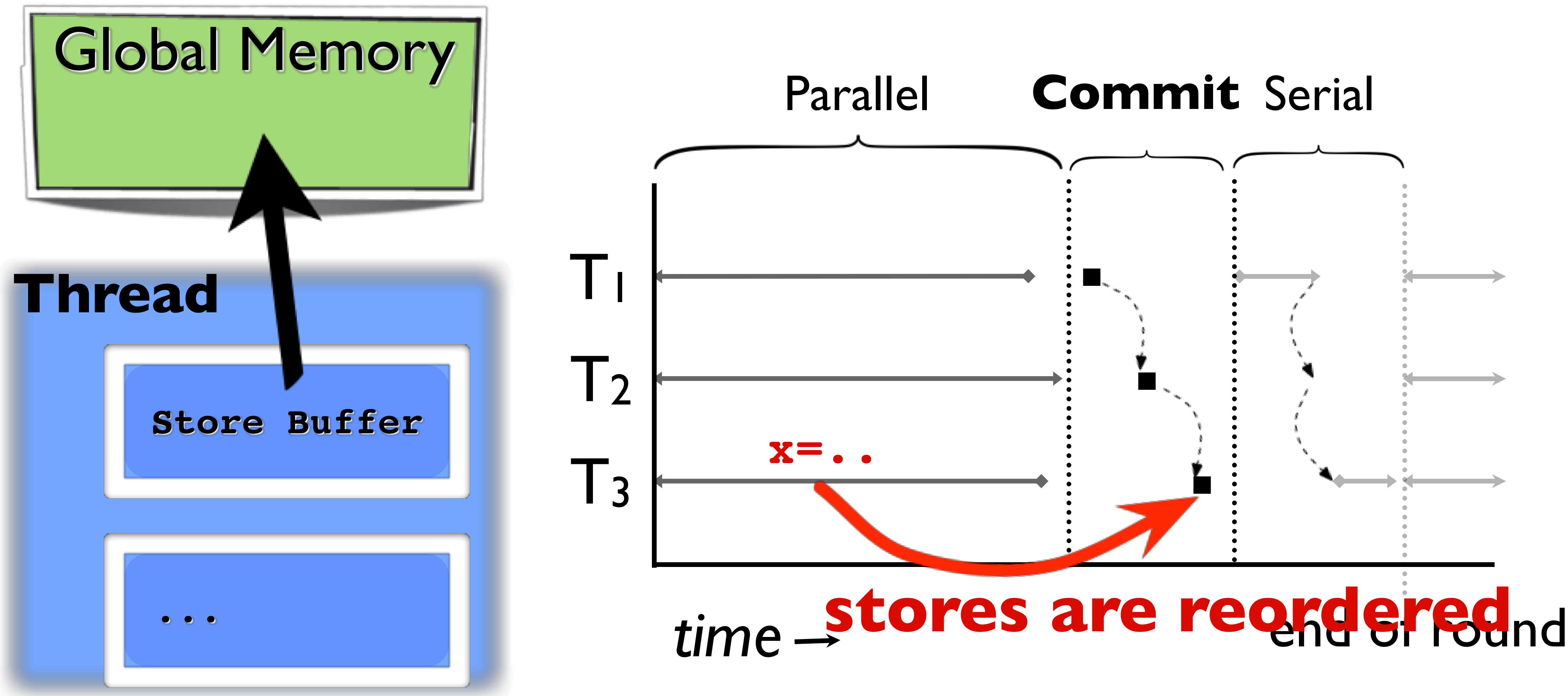
DMP-Buffering





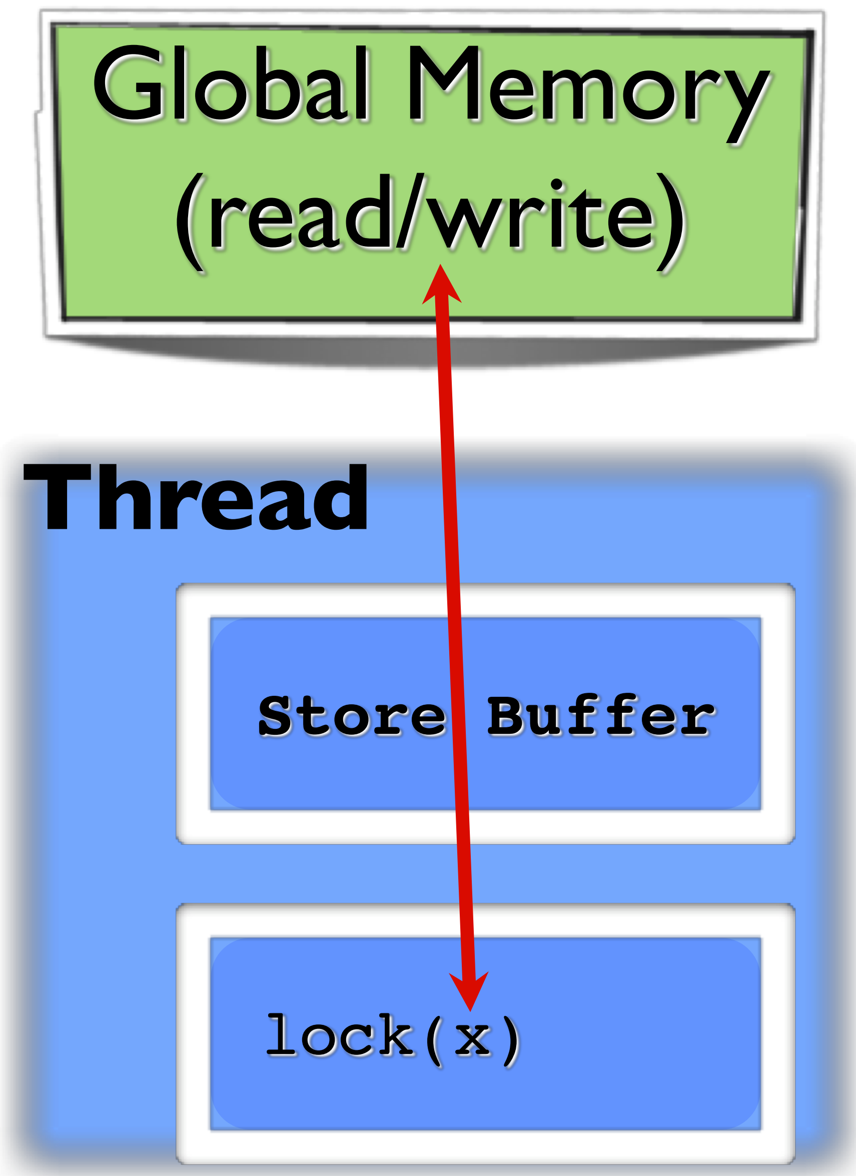
Parallel mode: buffer all stores (no communication)

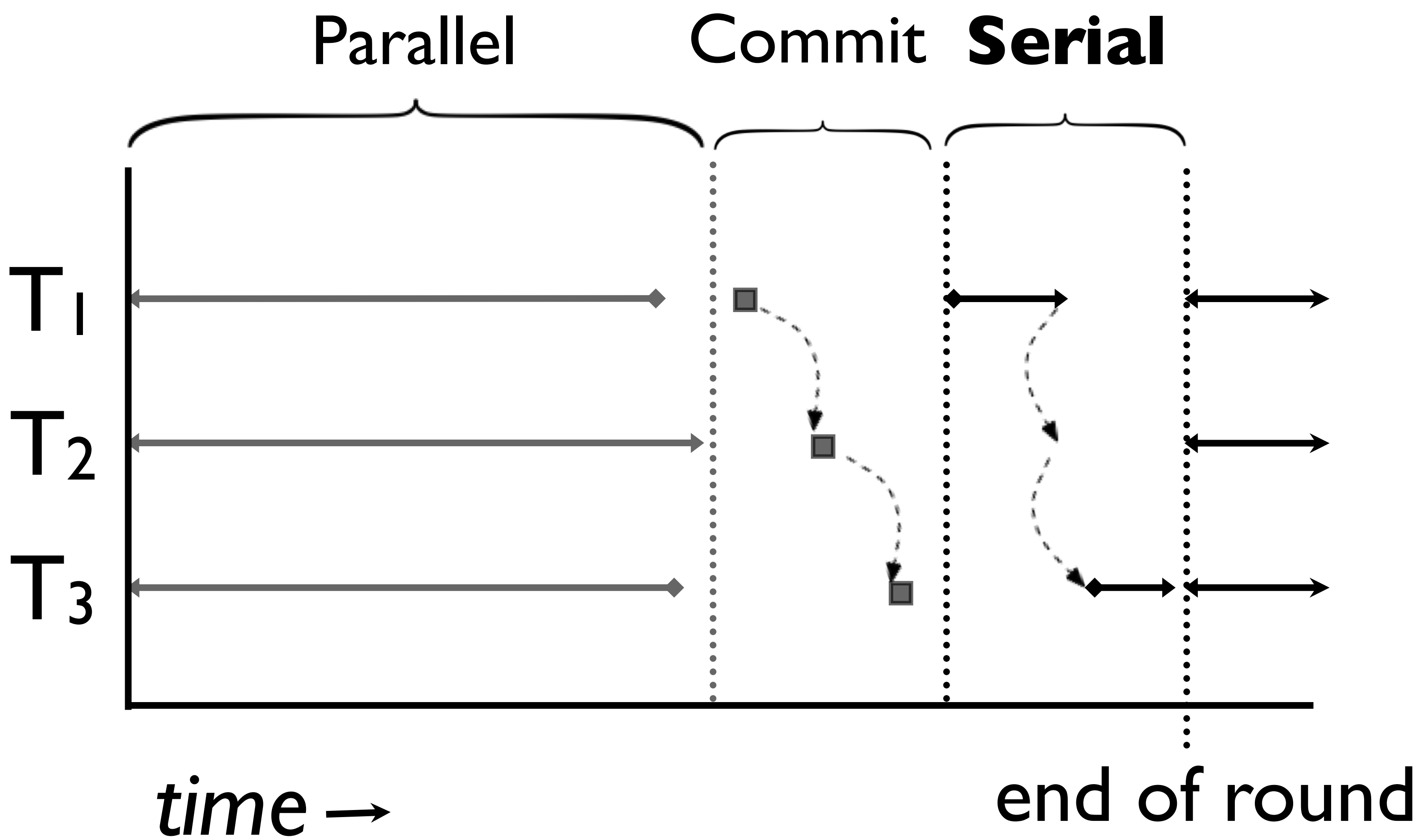
DMP-Buffering



Parallel mode: buffer all stores (no communication)
Commit mode: deterministically publish store buffers

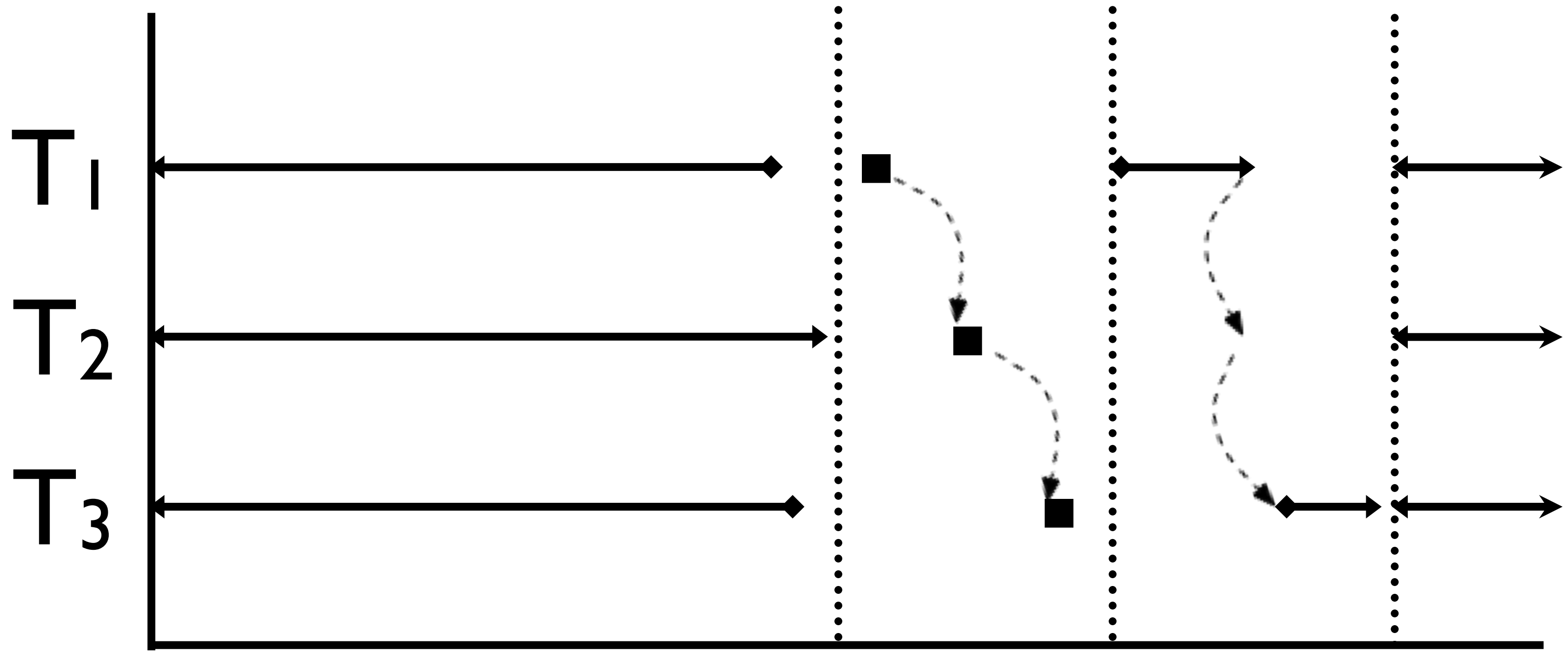
DMP-Buffering





Parallel mode: buffer all stores (no communication)
Commit mode: deterministically publish store buffers
Serial mode: used for synchronization (e.g. atomic ops)

DMP-Buffering



Parallel mode: buffer stores locally

- ends at *synchronization* (*atomic ops* and *fences*), and *quantum boundaries*

Commit mode: publish local store buffers

- logically serial for determinism
- executes in parallel for performance

Serial mode: used for synchronization (e.g. atomic ops)

DMP-Buffering

Thread 1

```
A = 1
if (B == 0)
   ...
```

Thread 2

```
B = 1
if (A == 0)
   ...
```

Dekker's Algorithm
(there is a data race)

DMP-Buffering

Thread 1

```
buffer[A] = 1
if (B == 0)
   ...
```

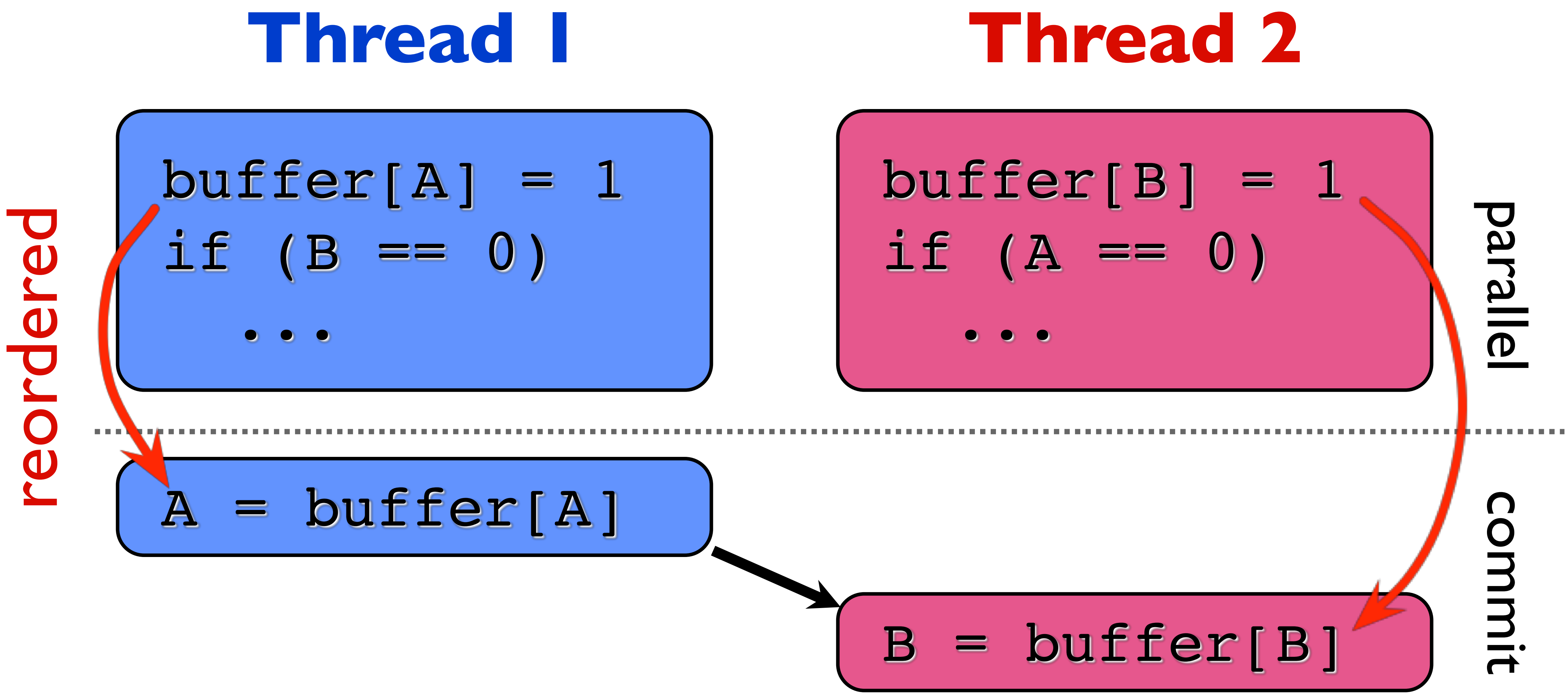
```
A = buffer[A]
```

Thread 2

```
buffer[B] = 1
if (A == 0)
   ...
```

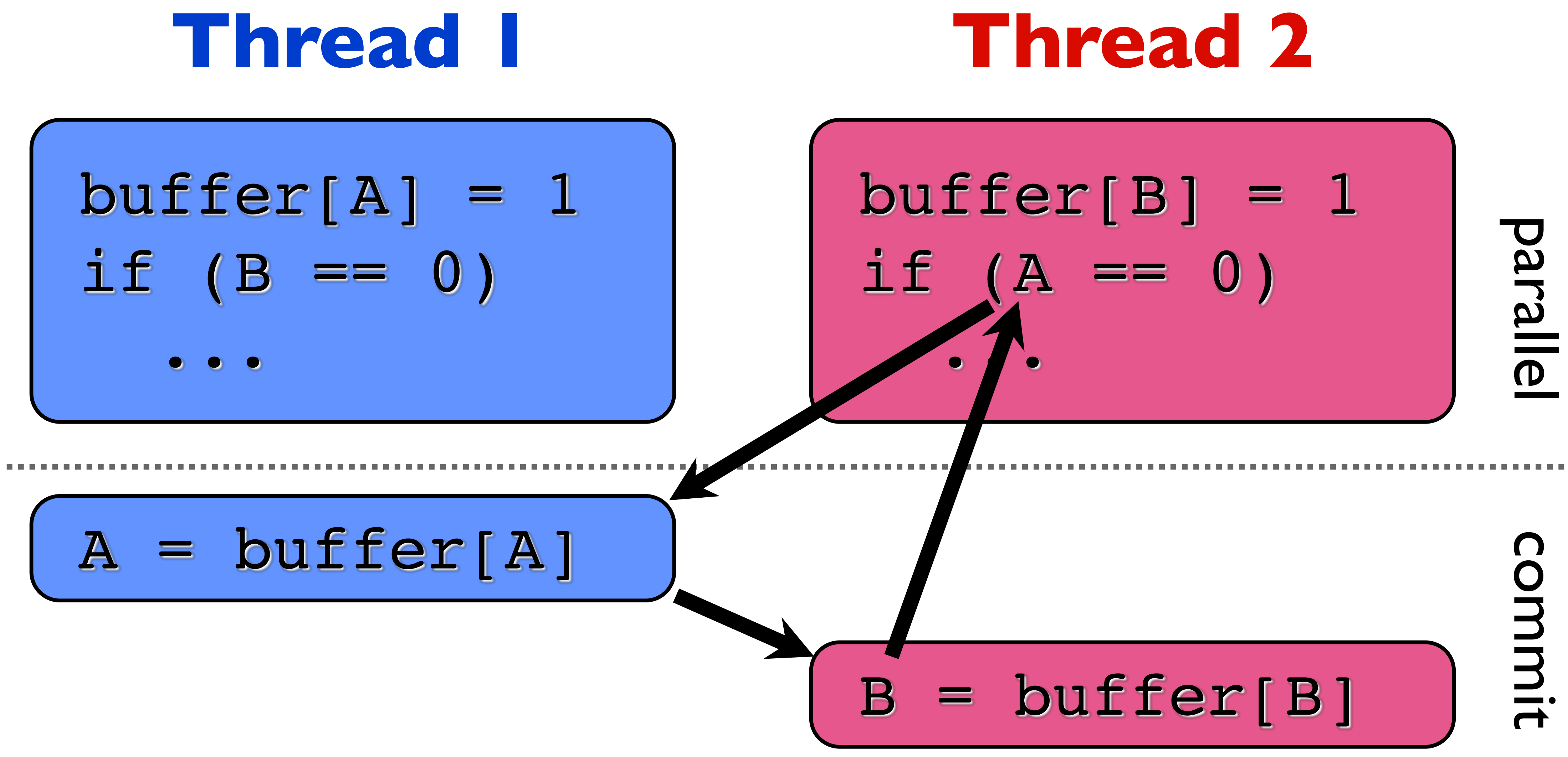
```
B = buffer[B]
```

DMP-Buffering



This is deterministic . . .

DMP-Buffering



. . . but not sequentially consistent
(cycle in the happens-before graph)

DMP-Buffering

Thread 1

```
A = 1
tmp1 = B
```

```
if (tmp1 == 0)
   ...
```

Thread 2

```
B = 1
tmp2 = A
```

```
if (tmp2 == 0)
   ...
```

Dekker's Algorithm (again)

Let's remove the data race . . .

DMP-Buffering

Thread 1

```
lock(L)

A = 1
tmp1 = B

unlock(L)

if (tmp1 == 0)
   ...
```

Thread 2

```
lock(L)

B = 1
tmp2 = A

unlock(L)

if (tmp2 == 0)
   ...
```

Dekker's Algorithm
(no data race)

DMP-Buffering

```
lock(L)
```

serial

```
A = 1
tmp1 = B
```

parallel + commit

```
unlock(L)
```

Synchronization happens sequentially

serial

```
lock(L)
```

```
if (tmp1 == 0)
   ...
```

```
B = 1
tmp2 = A
```

parallel + commit

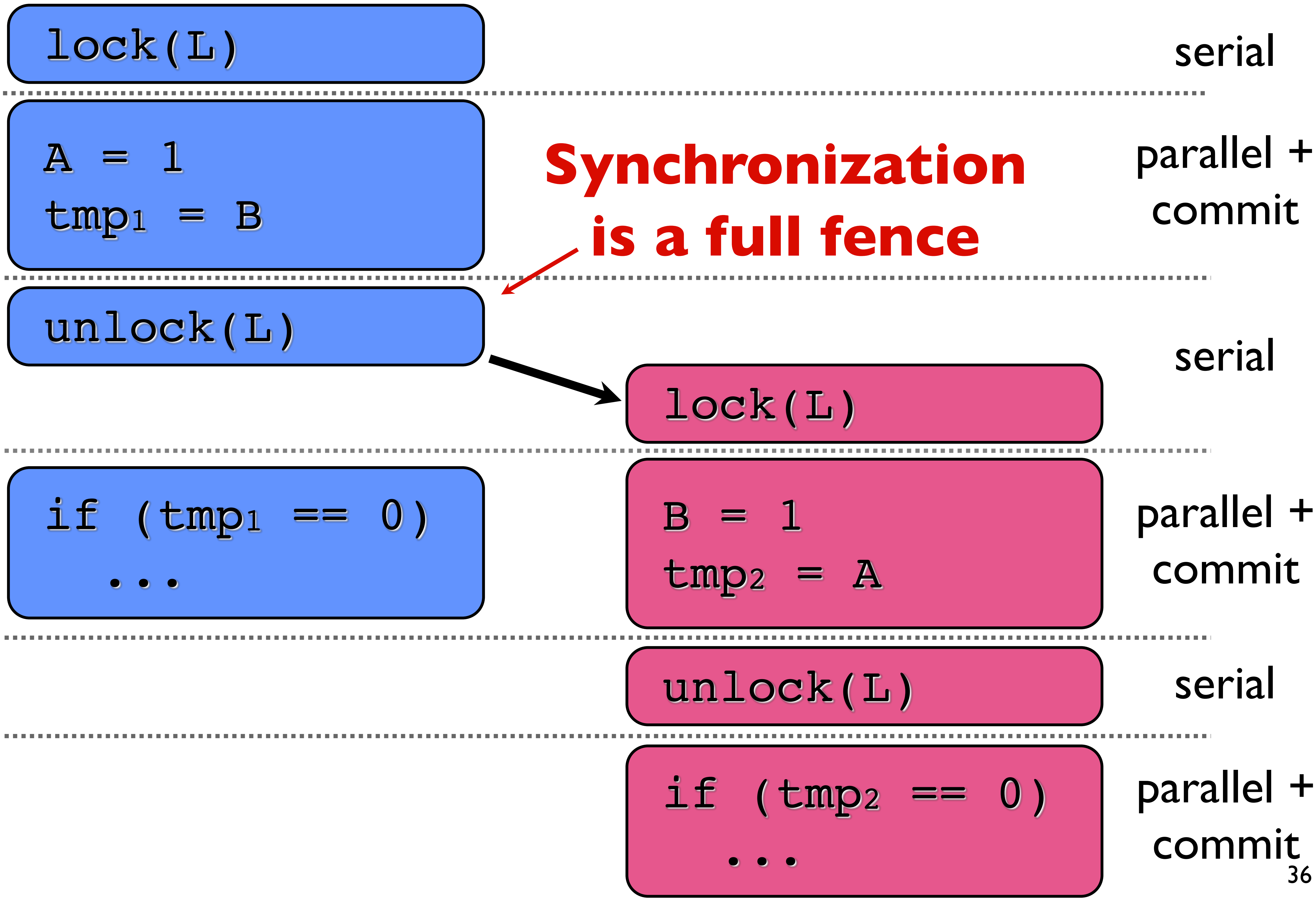
```
unlock(L)
```

serial

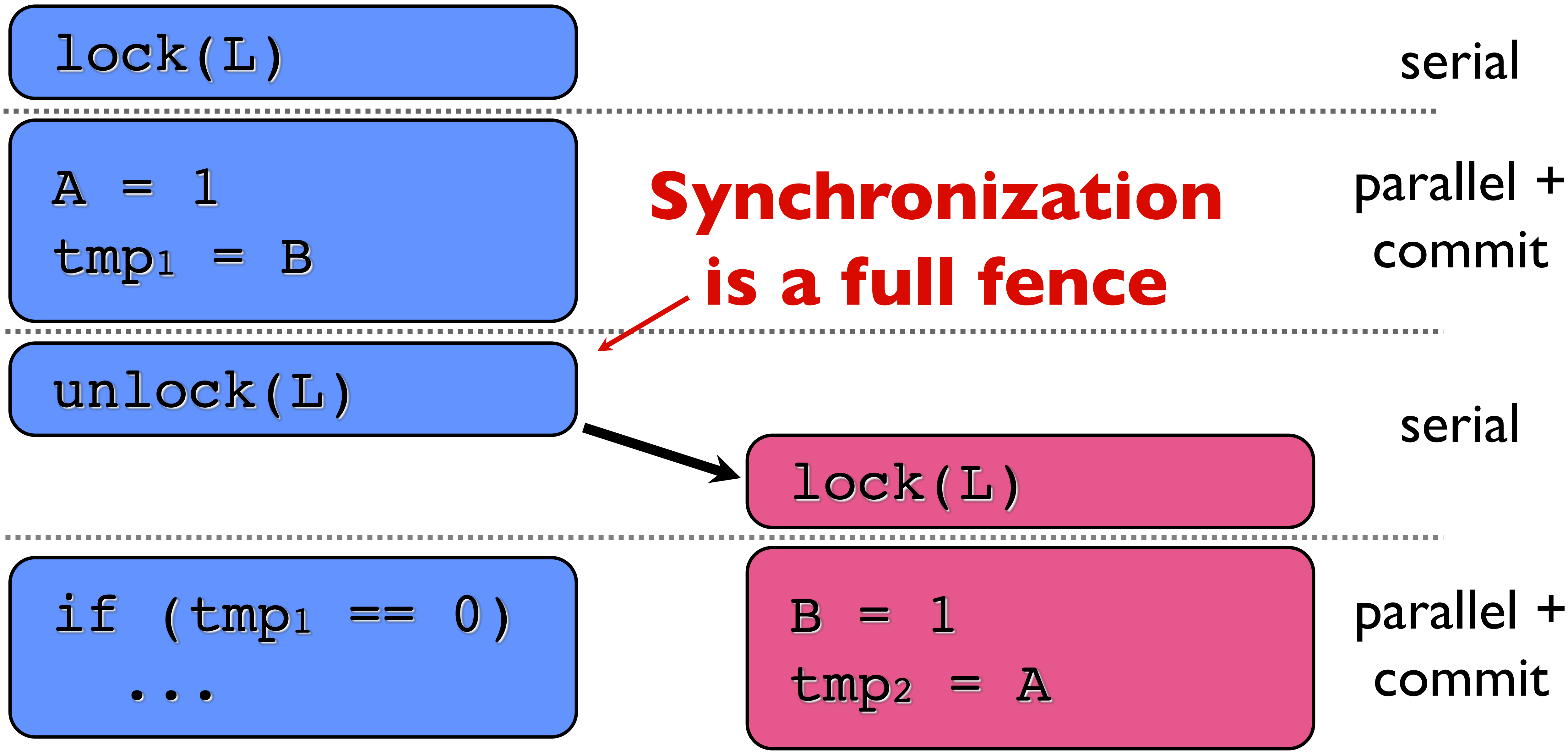
```
if (tmp2 == 0)
   ...
```

parallel + commit

DMP-Buffering



DMP-Buffering



Data race free programs are sequentially consistent
(required by C++ and Java memory models)

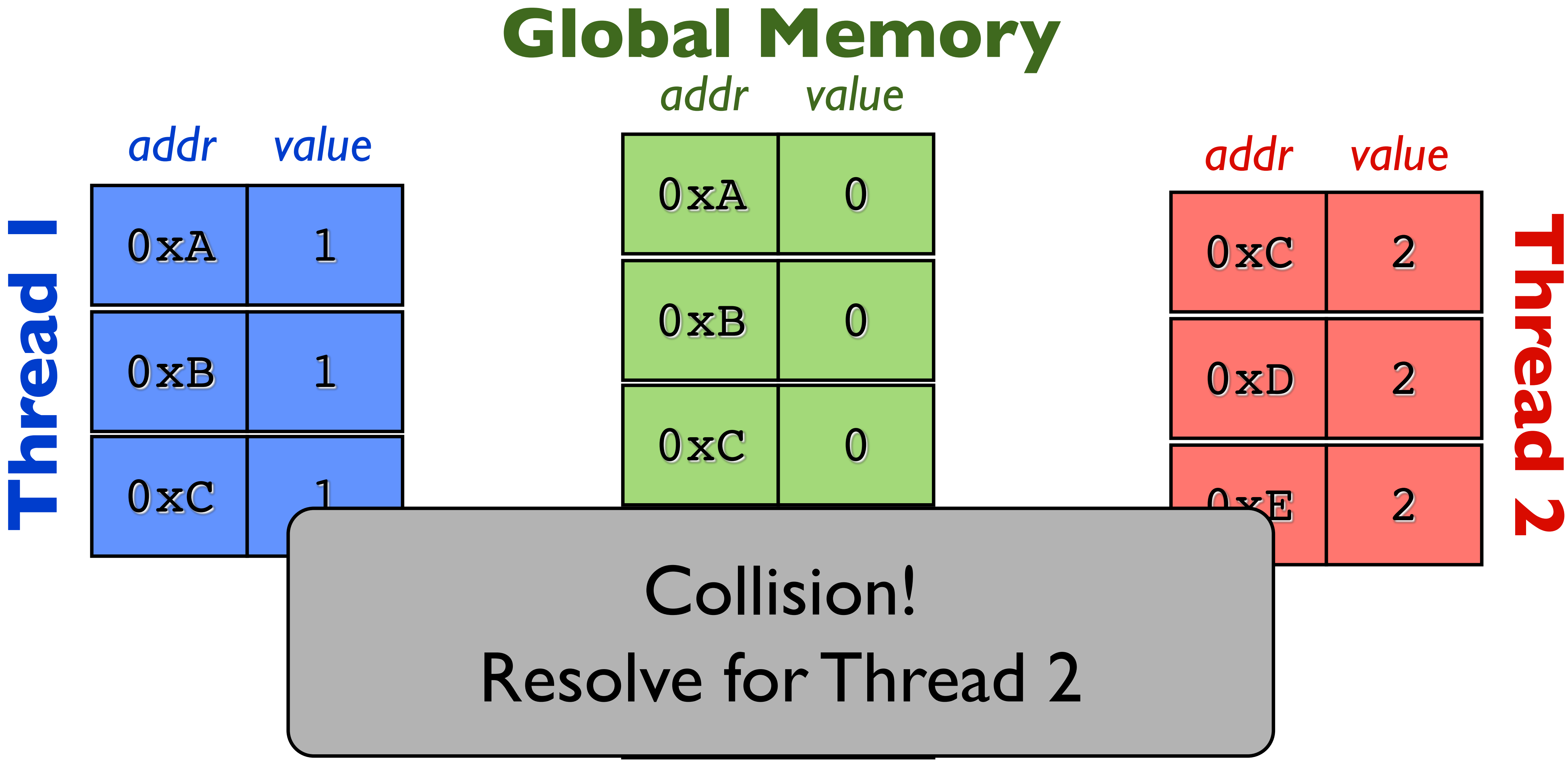
DMP-Buffering: Parallel Commit

For determinism, the commit order must be deterministic
i.e. logically serial

For performance, the commit must happen in parallel

Basic idea:
- Publish store buffers in parallel
- Preserve the commit order on collisions

DMP-Buffering: Parallel Commit



Basic idea:

- Publish store buffers in parallel
- Preserve the commit order on collisions

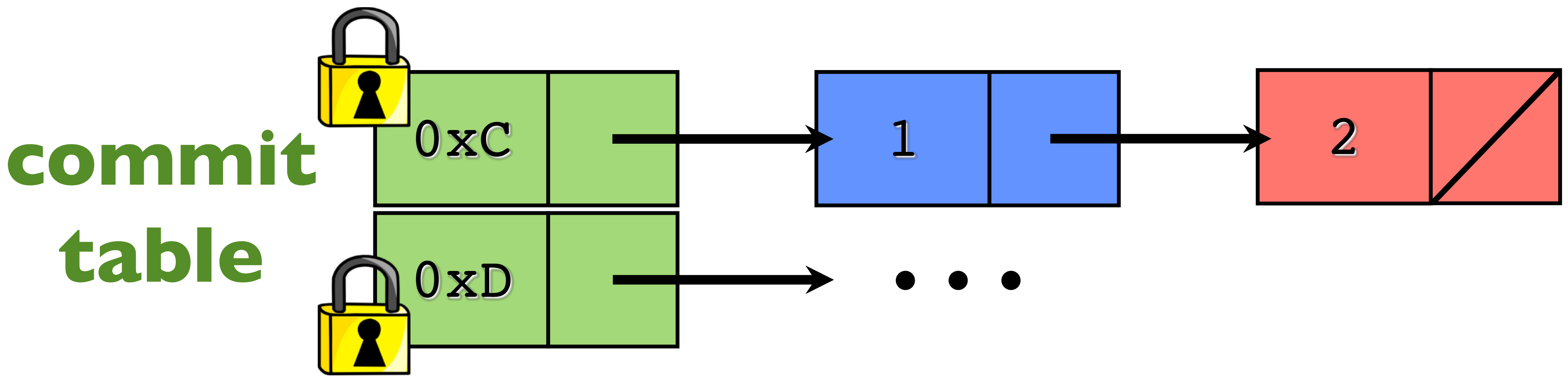
DMP-Buffering: Parallel Commit

Basic idea:

- Publish store buffers in parallel
- Preserve the commit order on collisions

Detecting collisions

- keep global record of published locations
- locks to serialize writes



- bloom filter to reduce locking overhead

Outline

Recap of DMP [ASPLOS'09]:

- DMP-Ownership
- DMP-TM
- What's wrong with DMP in software?

CoreDet:

- DMP-Buffering
- **Performance Evaluation**

What We Evaluated

Three algorithms implemented in CoreDet:

DMP-Ownership

DMP-Buffering

DMP-PartialBuffering
- a hybrid of DMP-Ownership and DMP-Buffering
- decides dynamically which data to buffer

Experimental Methodology

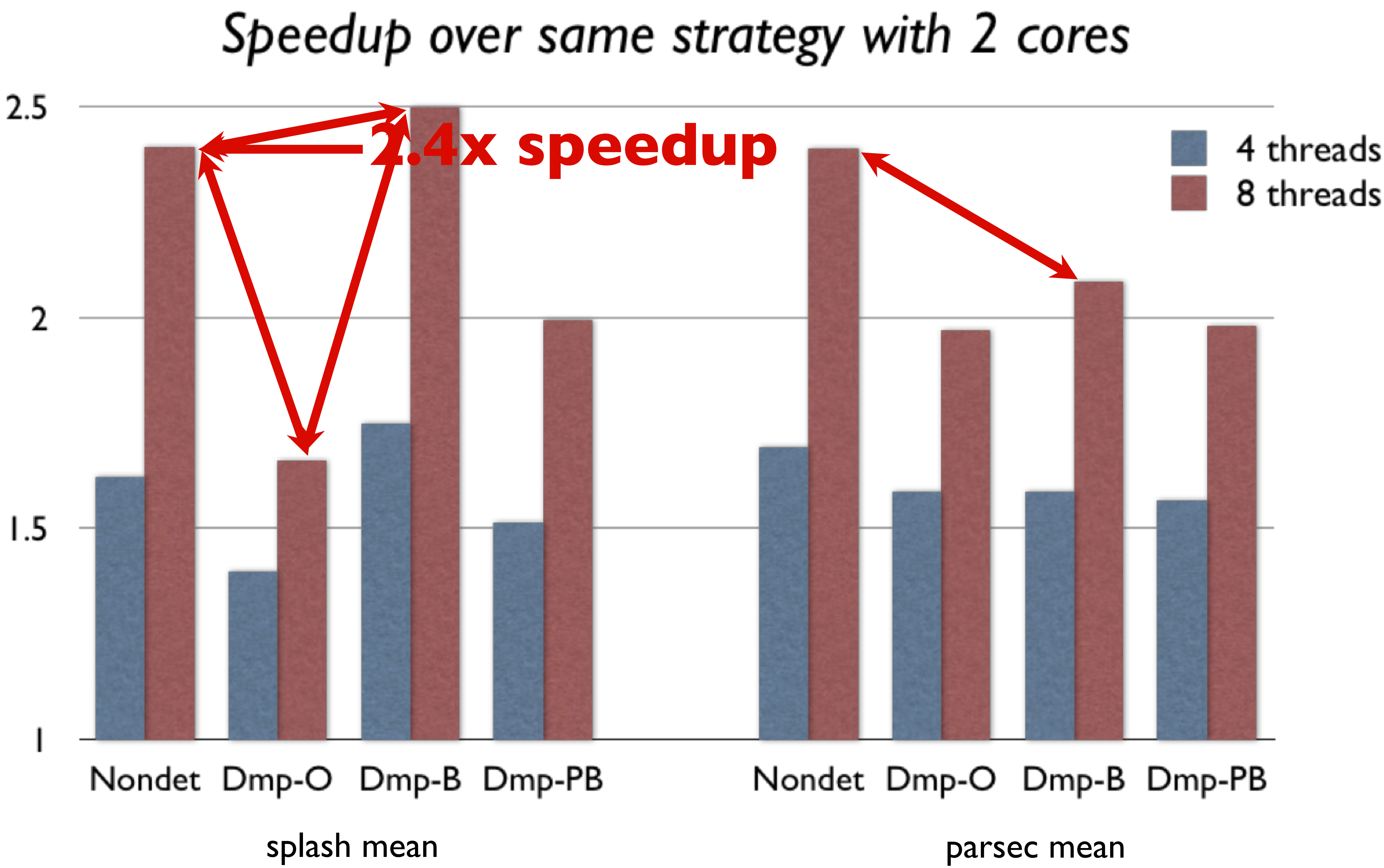
PARSEC and SPLASH2 benchmark suites

- 8-core Intel Xeon
- scaled inputs to run for about a minute

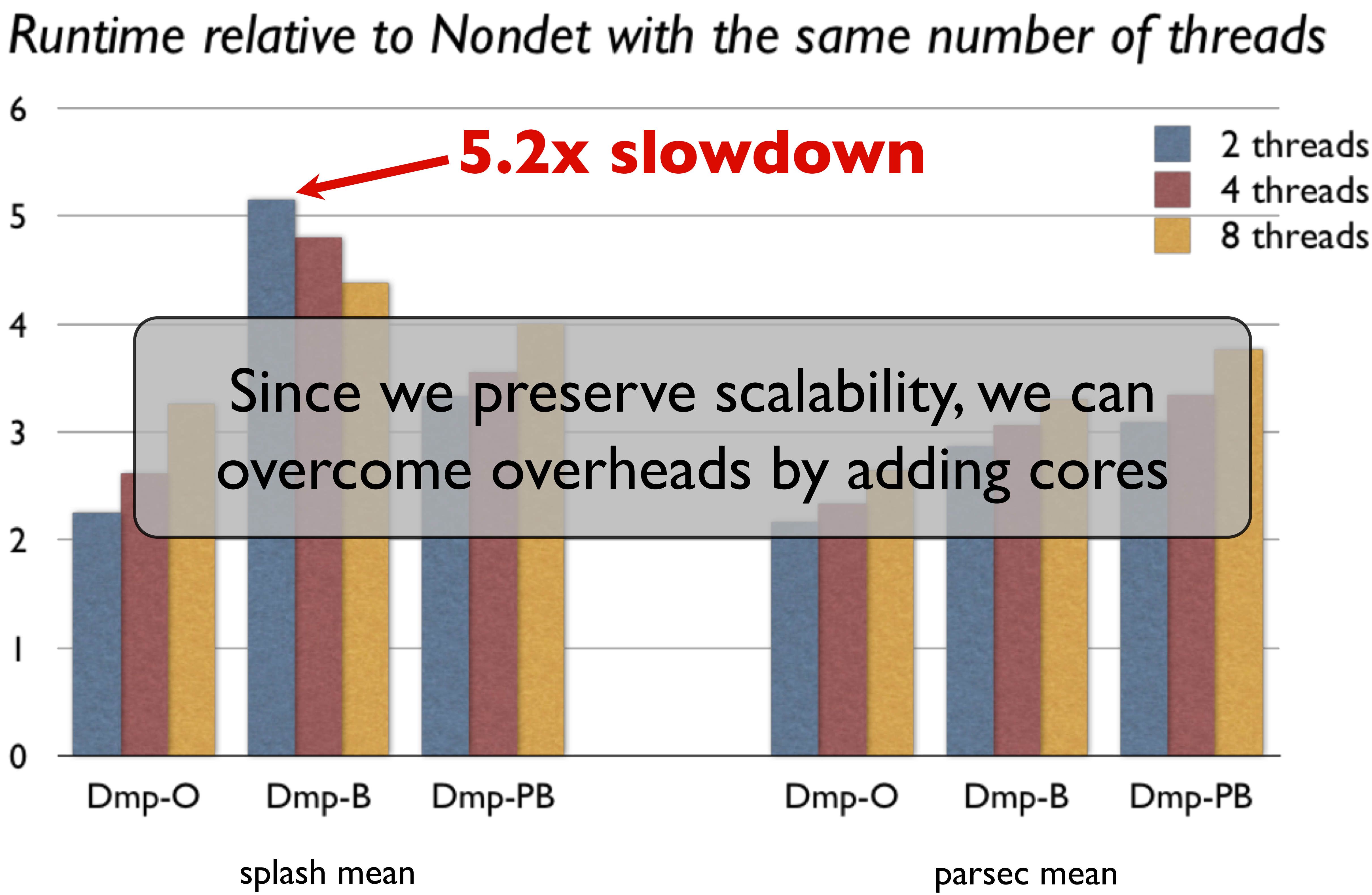
Goal: in comparison to nondeterministic execution ...

- What is the scalability?
- What are the overheads?

Scalability



Overheads



Runtime relative to Nondet with the same number of threads
6
5
4
3
2
1
0
5.2x slowdown
2 threads
4 threads
8 threads
Since we preserve scalability, we can overcome overheads by adding cores
Dmp-O
Dmp-B
Dmp-PB
Dmp-O
Dmp-B
Dmp-PB
splash mean
parsec mean

Wrap Up

CoreDet

- guarantees determinism in software of arbitrary C/C++ multithreaded programs

DMP-Buffering

- uses a relaxed memory consistency model
- scales comparably to nondeterministic execution

Also in the paper . . .

Compiler details

- static optimizations
- forming balanced quanta

Runtime details

- dealing with external libraries
- threading libraries
- memory allocation

Evaluation

- more detailed performance characterization

Thank you!

Questions?

the CoreDet source code is available at
http://sampa.cs.washington.edu

(backup slides)

DMP-Buffering

Atomic ops must happen in serial mode

`CAS(X,a,b)`

`CAS(X,a,c)`

DMP-Buffering

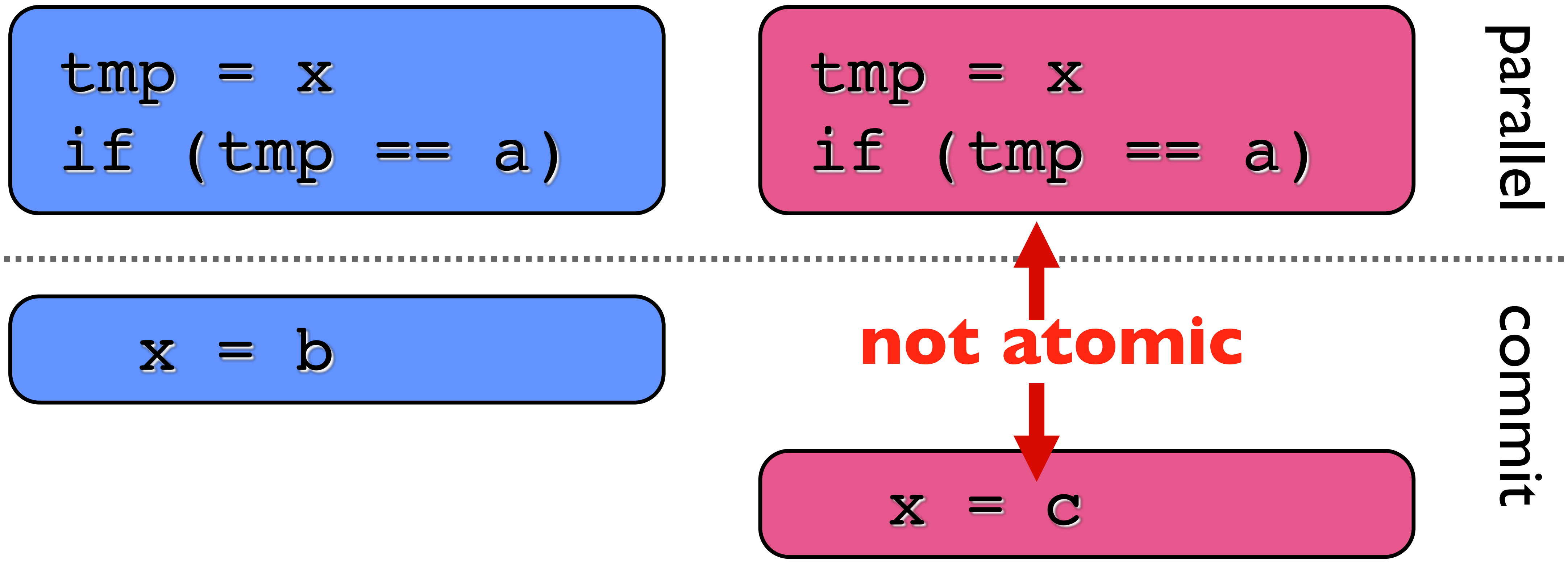
Atomic ops must happen in serial mode

```
tmp = x
if (tmp == a)
  x = b
```

```
tmp = x
if (tmp == a)
  x = c
```

DMP-Buffering

Atomic ops must happen in serial mode



Synchronization, e.g. `lock()`, must happen in serial mode

- These are atomic ops
- There is an implied fence (must flush store buffers)

CoreDet: Implementation

A compiler (LLVM pass)

- instruments the code with calls to the runtime
- static optimizations to remove instrumentation
  - escape analysis
  - redundancy analysis

A runtime library

- scheduling threads
- tracks interthread communication
- deterministic wrappers for . . .
  - pthreads
  - malloc

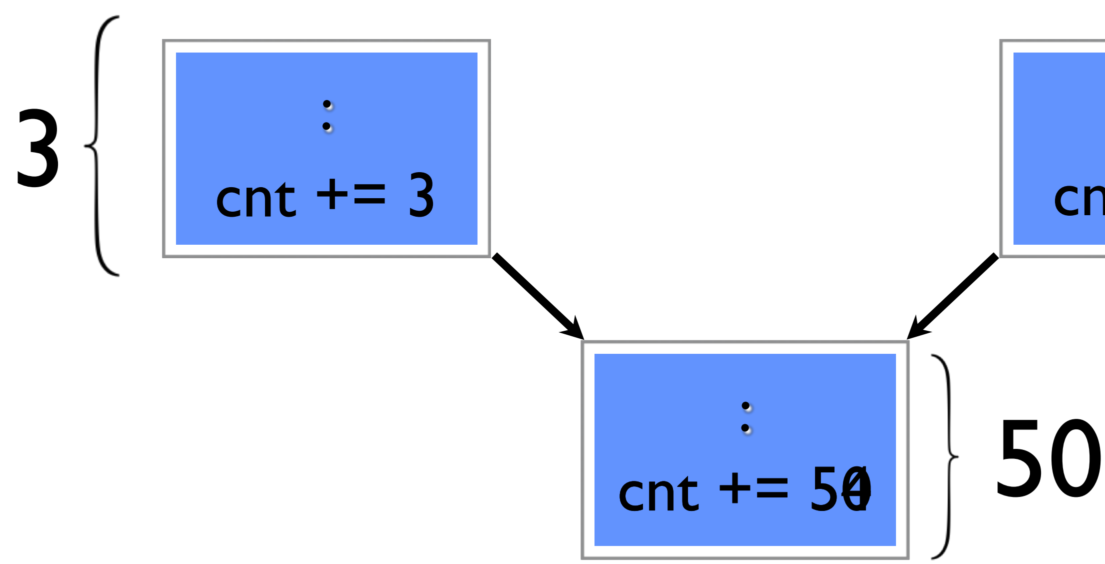
Quantum Formation

"Just" instruction counting

Tension between:

- Perfect counting, for maximal balance
  - e.g. every basic block
- Minimal counting, for minimal overhead
  - e.g. only backedges and recursive calls

Heuristic compromise:



Remove Instrumention From ...

- Accesses to thread-local (non-escaping) objects
- Redundant accesses

```
y = ... x ...
z = ... x ...
```

don’t need to instrument this

Remove Instrumention From ...

- Accesses to thread-local (non-escaping) objects

  DMP-Buffering: requires unification-based points-to analysis

```
int local;
int *p = (...) ? &local : &global;
...
```

must access through the store buffer

- Redundant accesses

```
y = ... x ...
z = ... x ...
```

don't need to instrument this

Remove Instrumention From ...

- Accesses to thread-local (non-escaping) objects

  DMP-Buffering: requires unification-based points-to analysis

```
int local;
int *p = (...) ? &local : &global;
...
```

must access through the store buffer

- Redundant accesses

```
y = ... x ...
z = ... x ...
```

don't need to instrument this

Remove Instrumention From ...

- Accesses to thread-local (non-escaping) objects

  DMP-Buffering: requires unification-based points-to analysis

```
int local;
int *p = (...) ? &local : &global;
...
```

  must access through the store buffer

- Redundant accesses

```
y = ... x ...
z = ... x ...
```

  don't need to instrument this

  DMP-Buffering: this does not apply

External Libraries

We do not instrument external shared libraries, such as the system `libc`

1. External calls must be serialized

Preventing over-serialization:

- We check indirect calls at runtime
- We provide deterministic wrappers for common `libc` functions, e.g. `memcpy` and `malloc`
- We do not serialize pure `libc` functions, e.g. `sqrt`