

Lecture 12:

Memory Consistency

Parallel Computing
Stanford CS149, Fall 2019

What's Due

- **Nov 1**
 - **Assignment 3: A Simple Renderer in CUDA**
- **Nov 4**
 - **Written Assignment 3**
- **Nov 5**
 - **Midterm**
 - **Open book, open notes**
 - **Review session on Nov 3**

Shared Memory Behavior

- Intuition says loads should return latest value written
 - What is latest?
 - Coherence: only one memory location
 - Consistency: apparent ordering for all locations
 - Order in which memory operations performed by one thread become visible to other threads
- Affects
 - Programmability: how programmers reason about program behavior
 - Allowed behavior of multithreaded programs executing with shared memory
 - Performance: limits HW/SW optimizations that can be used
 - Reordering memory operations to hide latency

Today: what you should know

- Understand the motivation for relaxed consistency models
- Understand the implications of relaxing $W \rightarrow R$ ordering
- Understand how to program correctly with relaxed consistency

Today: who should care

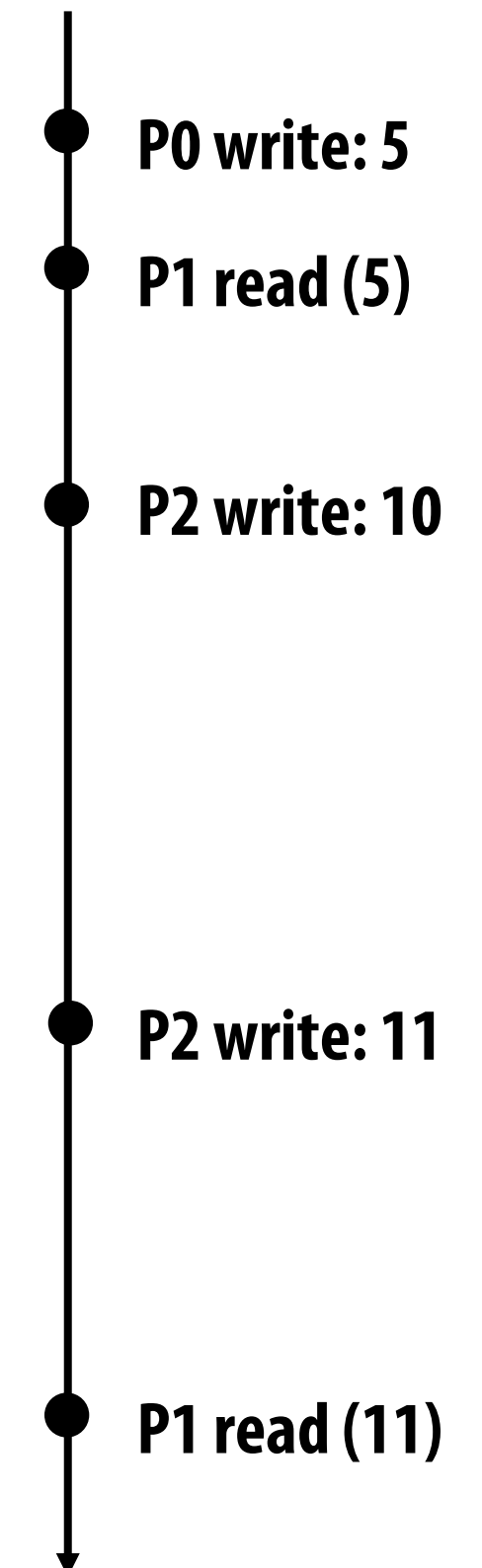
- Anyone who:
 - Wants to implement a synchronization library
 - Will ever work a job in kernel (or driver) development
 - Seeks to implement lock-free data structures *

* **Topic of a later lecture**

Memory coherence vs. memory consistency

- **Memory coherence** defines requirements for the observed behavior of reads and writes to the same memory location
 - All processors must agree on the order of reads/writes to X
 - In other words: it is possible to put all operations involving X on a timeline such that the observations of all processors are consistent with that timeline
- **Memory consistency** defines the behavior of reads and writes to different locations (as observed by other processors)
 - Coherence only guarantees that writes to address X will eventually propagate to other processors
 - Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

Observed chronology of operations on address X



Coherence vs. Consistency

(said again, perhaps more intuitively this time)

- The goal of cache coherence is to ensure that the memory system in a parallel computer behaves as if the caches were not there
 - Just like how the memory system in a uni-processor system behaves as if the cache was not there
- A system without caches would have no need for cache coherence
- Memory consistency defines the allowed behavior of loads and stores to different addresses in a parallel system
 - The allowed behavior of memory should be specified whether or not caches are present (and that's what a memory consistency model does)

Memory Consistency

- The trailer:
 - Multiprocessors reorder memory operations in unintuitive and strange ways
 - This behavior is required for performance
 - Application programmers rarely see this behavior
 - Systems (OS and compiler) developers see it all the time

Memory operation ordering

- A program defines a sequence of loads and stores (this is the “program order” of the loads and stores)

- Four types of memory operation orderings
 - $W_X \rightarrow R_Y$: write to X must commit before subsequent read from Y *
 - $R_X \rightarrow R_Y$: read from X must commit before subsequent read from Y
 - $R_X \rightarrow W_Y$: read to X must commit before subsequent write to Y
 - $W_X \rightarrow W_Y$: write to X must commit before subsequent write to Y

* To clarify: “write must commit before subsequent read” means:

When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs.

Multiprocessor Execution

Initially $A = B = 0$

Proc 0

(1) $A = 1$

(2) print B

Proc 1

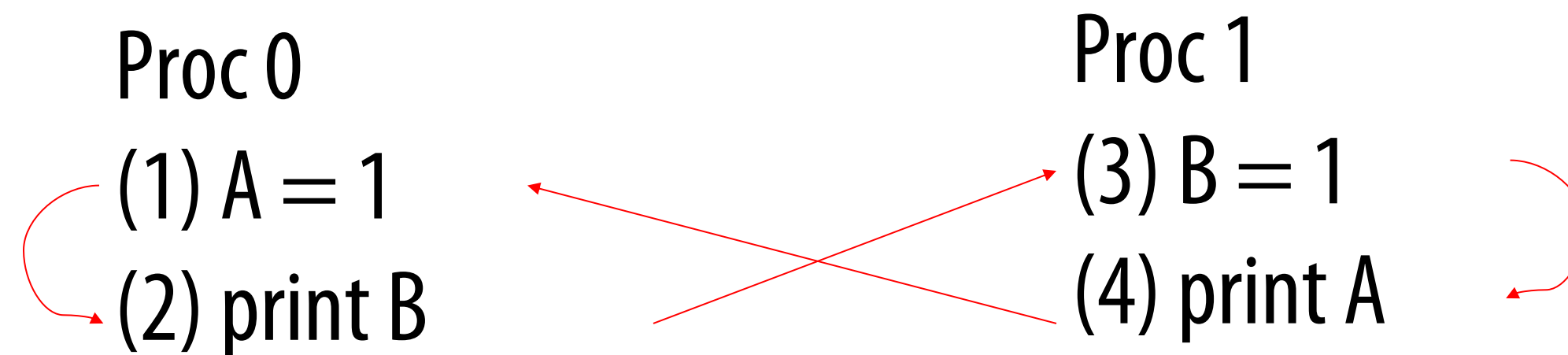
(3) $B = 1$

(4) print A

- What can be printed?
 - "01"?
 - "10"?
 - "11"?
 - "00"?

Orderings That Should Not Happen

Initially $A = B = 0$



- The program should not print “10” or “00”
- A “happens-before” graph shows the order in which events must execute to get a desired outcome
- If there’s a cycle in the graph, an outcome is impossible—an event must happen before itself!

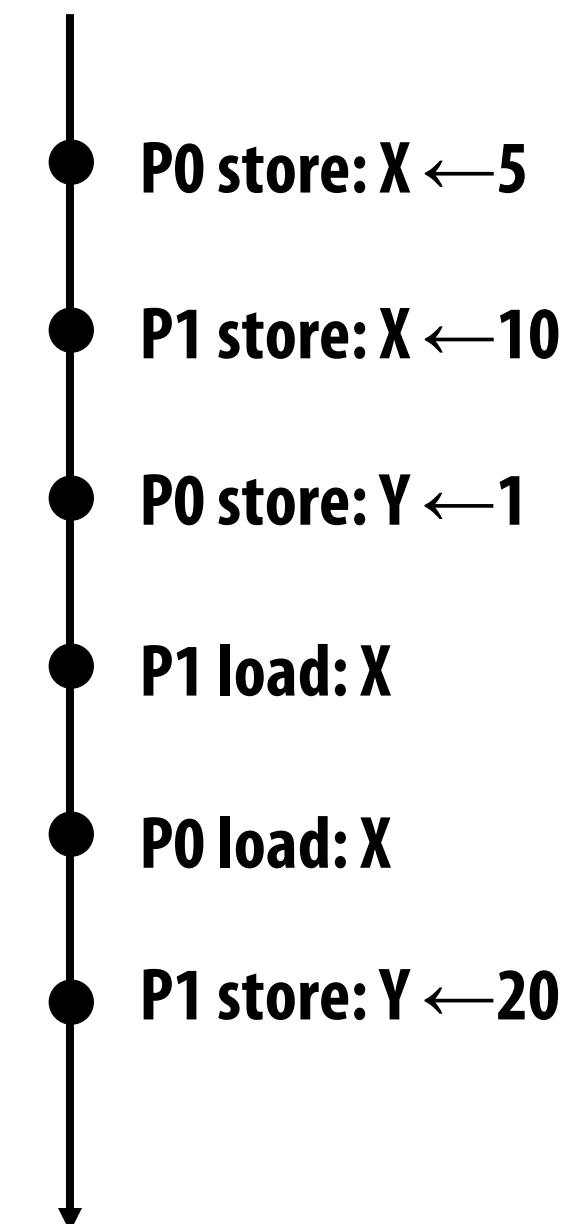
What Should Programmers Expect

■ Sequential Consistency

- Lamport 1976 (Turing Award 2013)
- All operations executed in some sequential order
 - As if they were manipulating a single shared memory
- Each thread's operations happen in program order

- A sequentially consistent memory system maintains all four memory operation orderings ($W_X \rightarrow R_Y, R_X \rightarrow R_Y, R_X \rightarrow W_Y, W_X \rightarrow W_Y$)

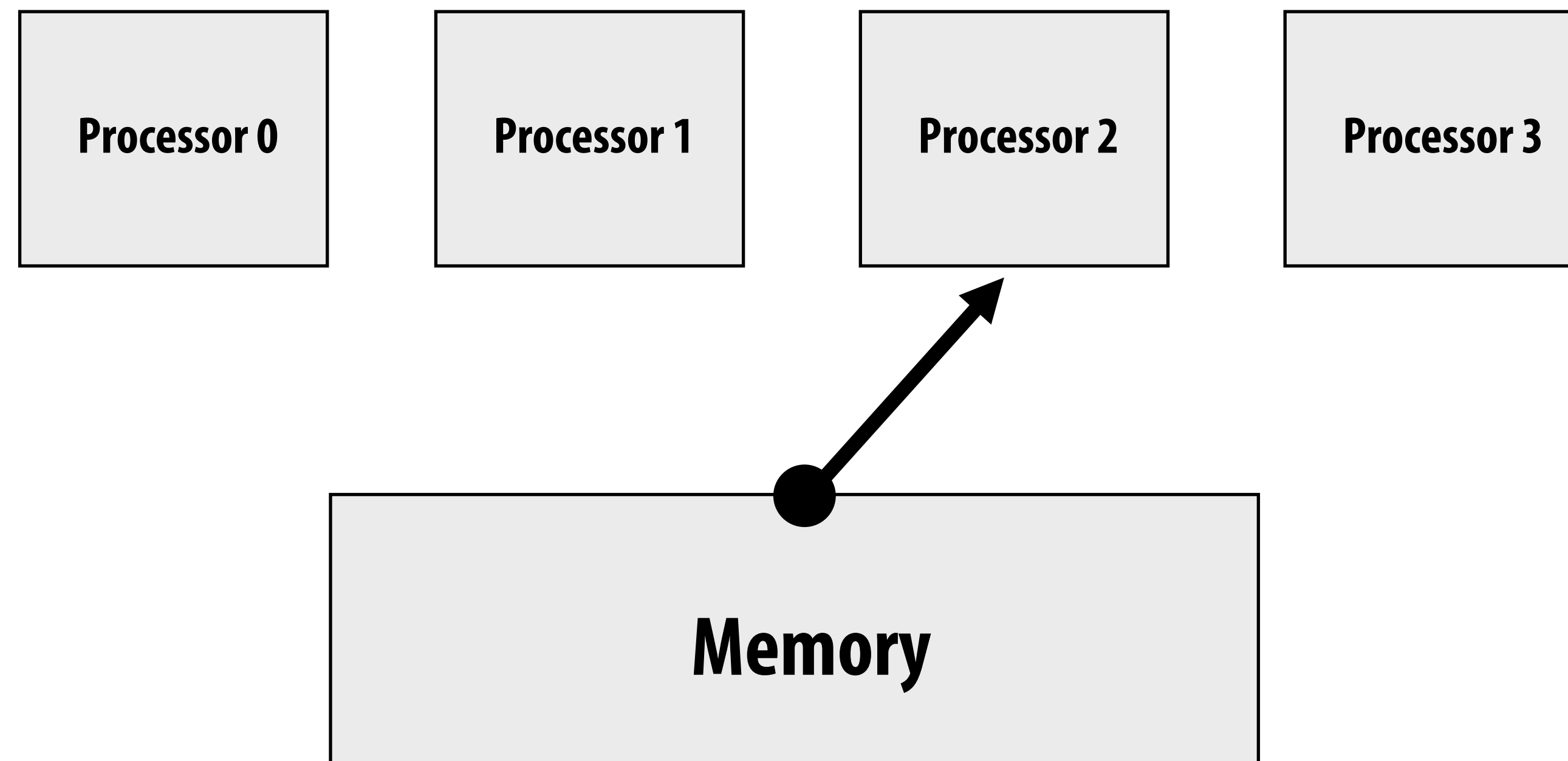
There is a chronology of all memory operations that is consistent with observed values



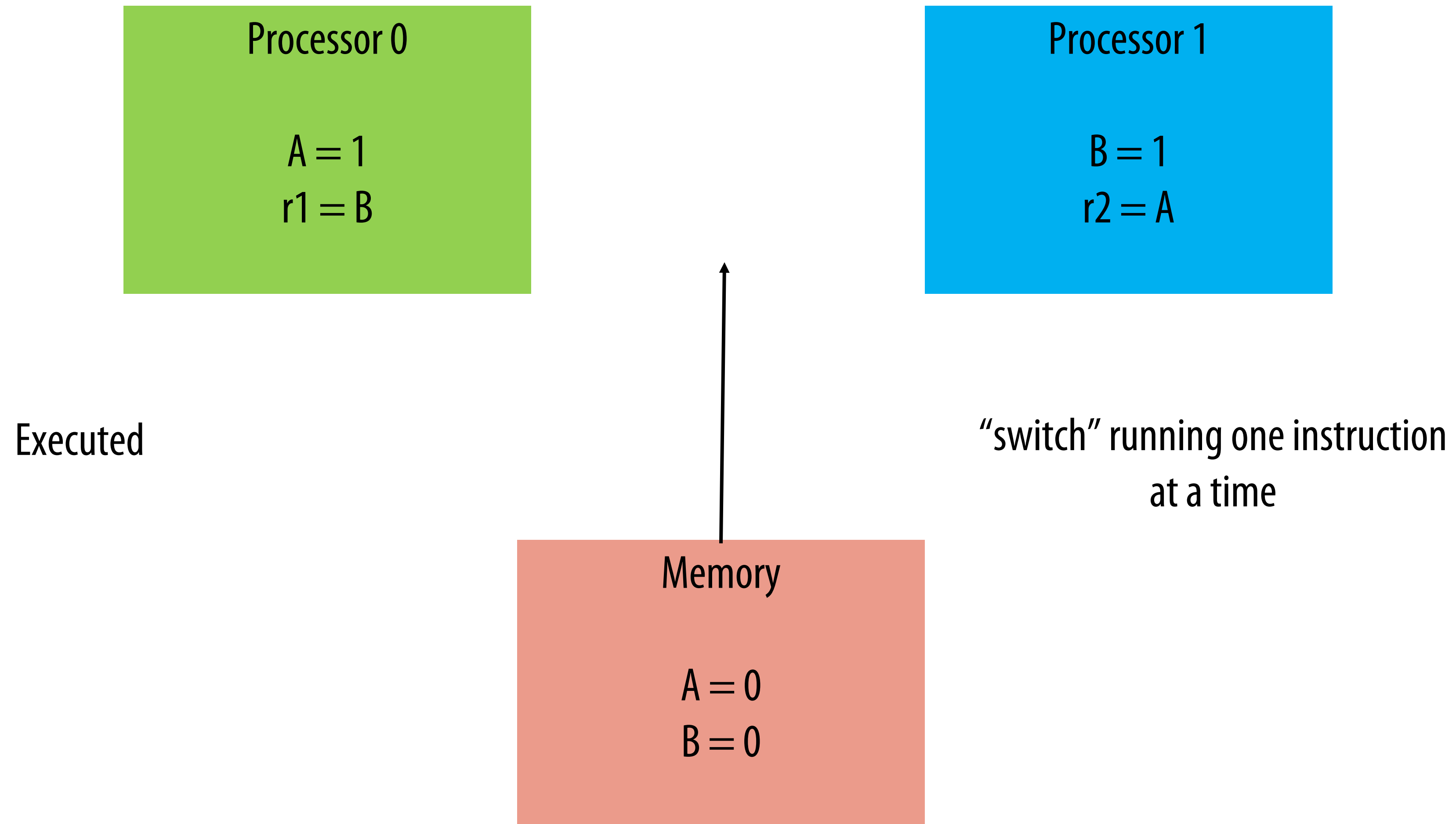
Note, now timeline lists operations to addresses X and Y

Sequential consistency (switch metaphor)

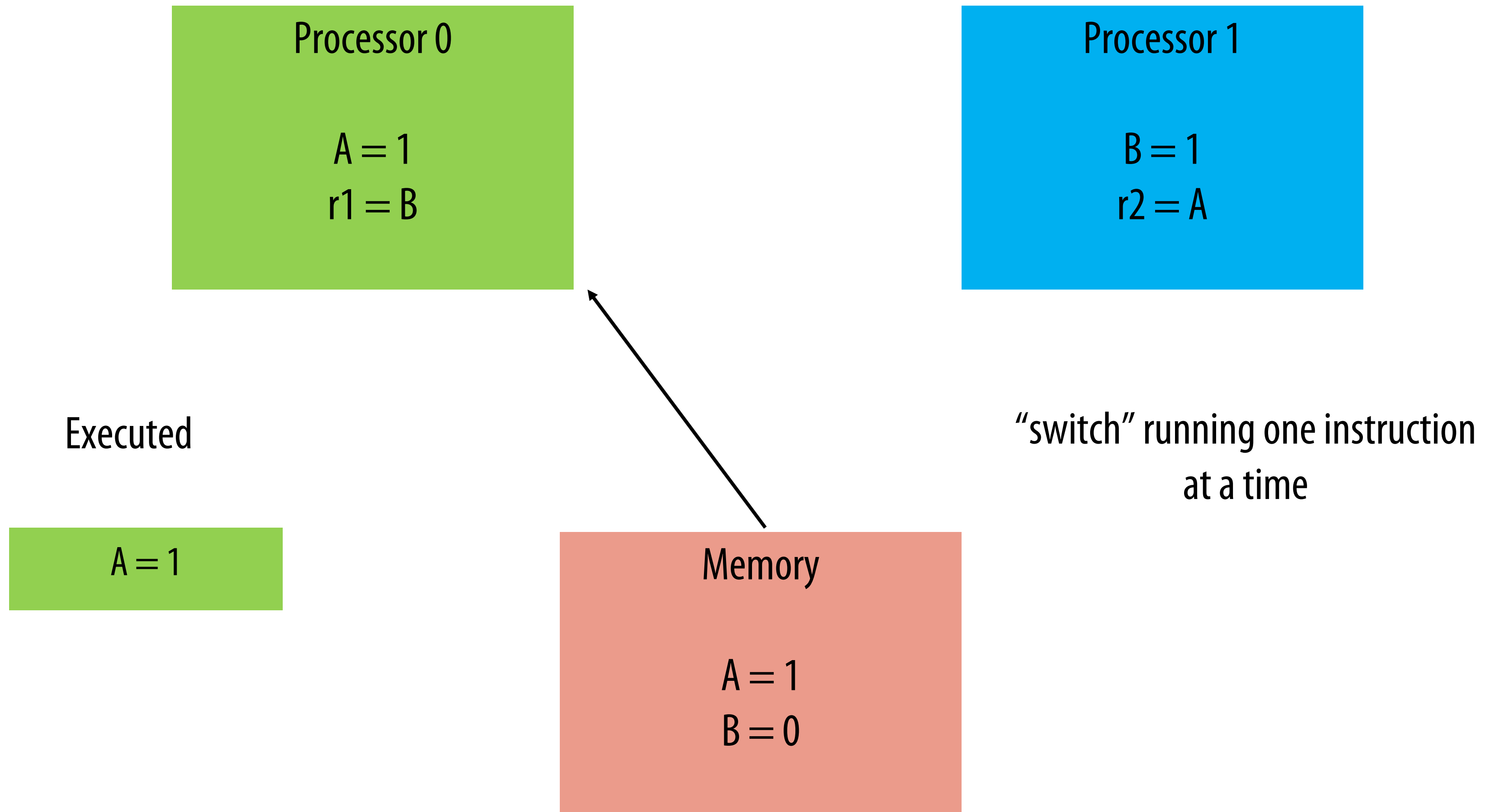
- All processors issue loads and stores in program order
- Memory chooses a processor at random, performs a memory operation to completion, then chooses another processor, ...



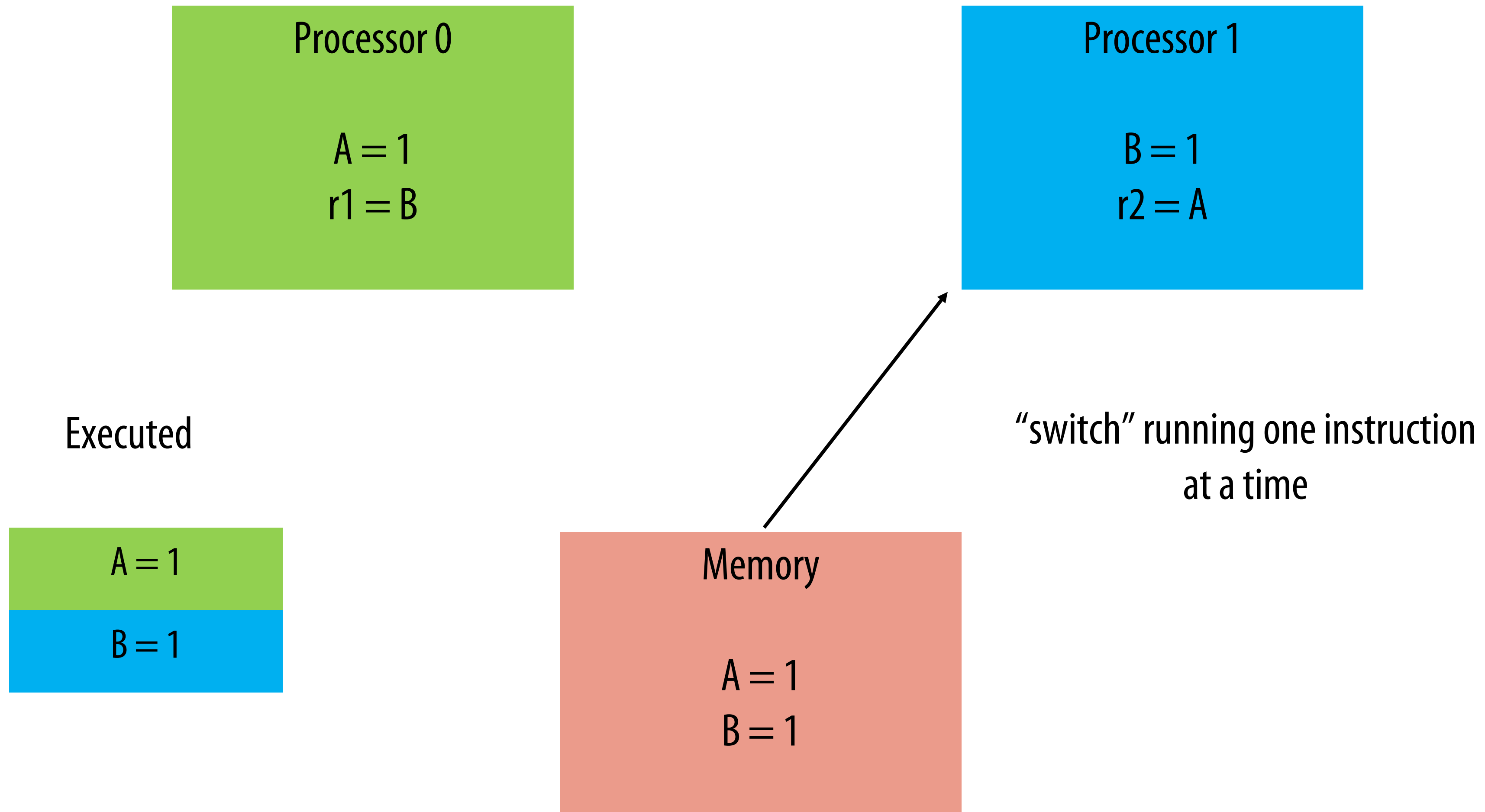
Sequential Consistency Example



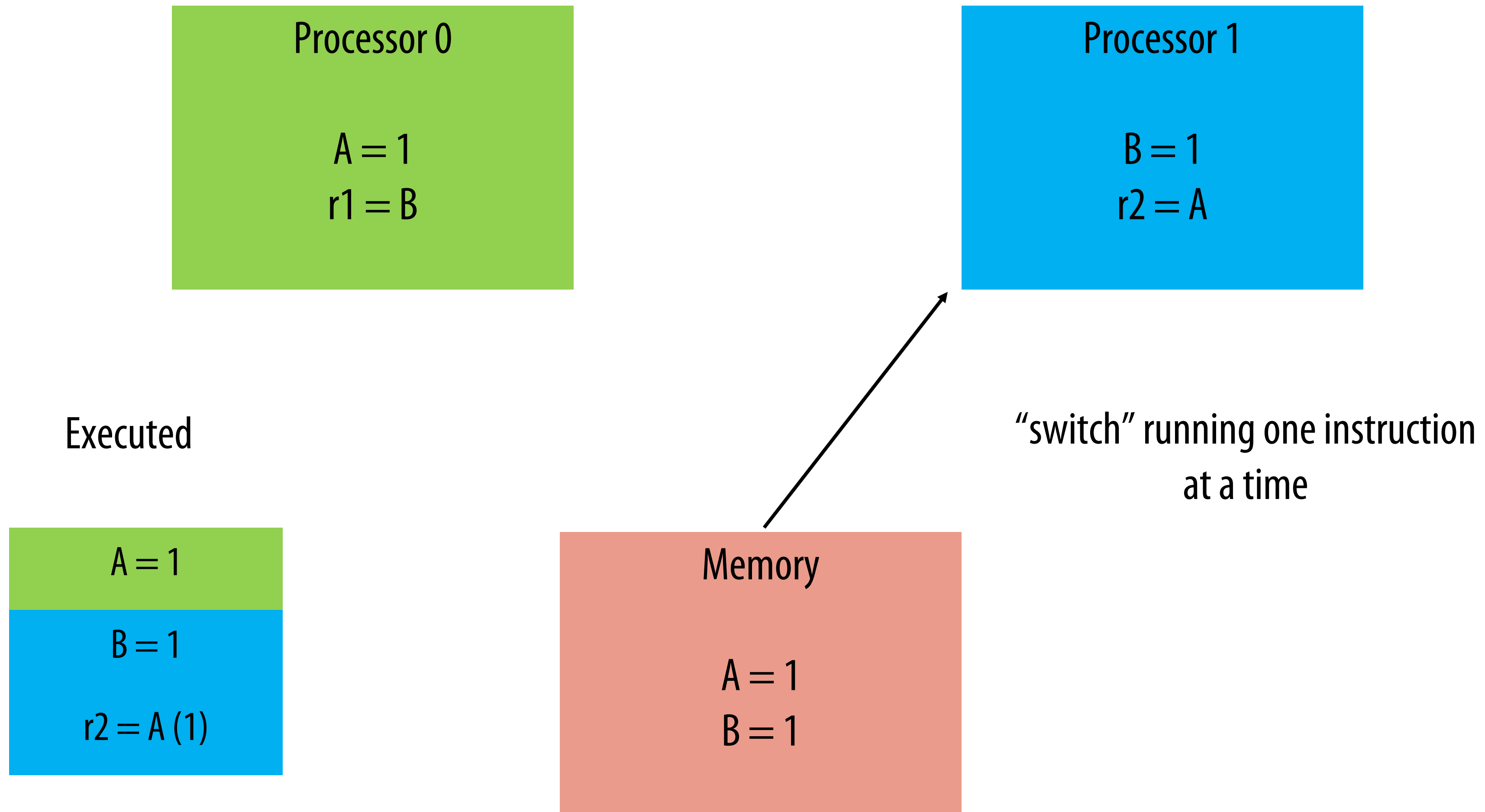
Sequential Consistency Example



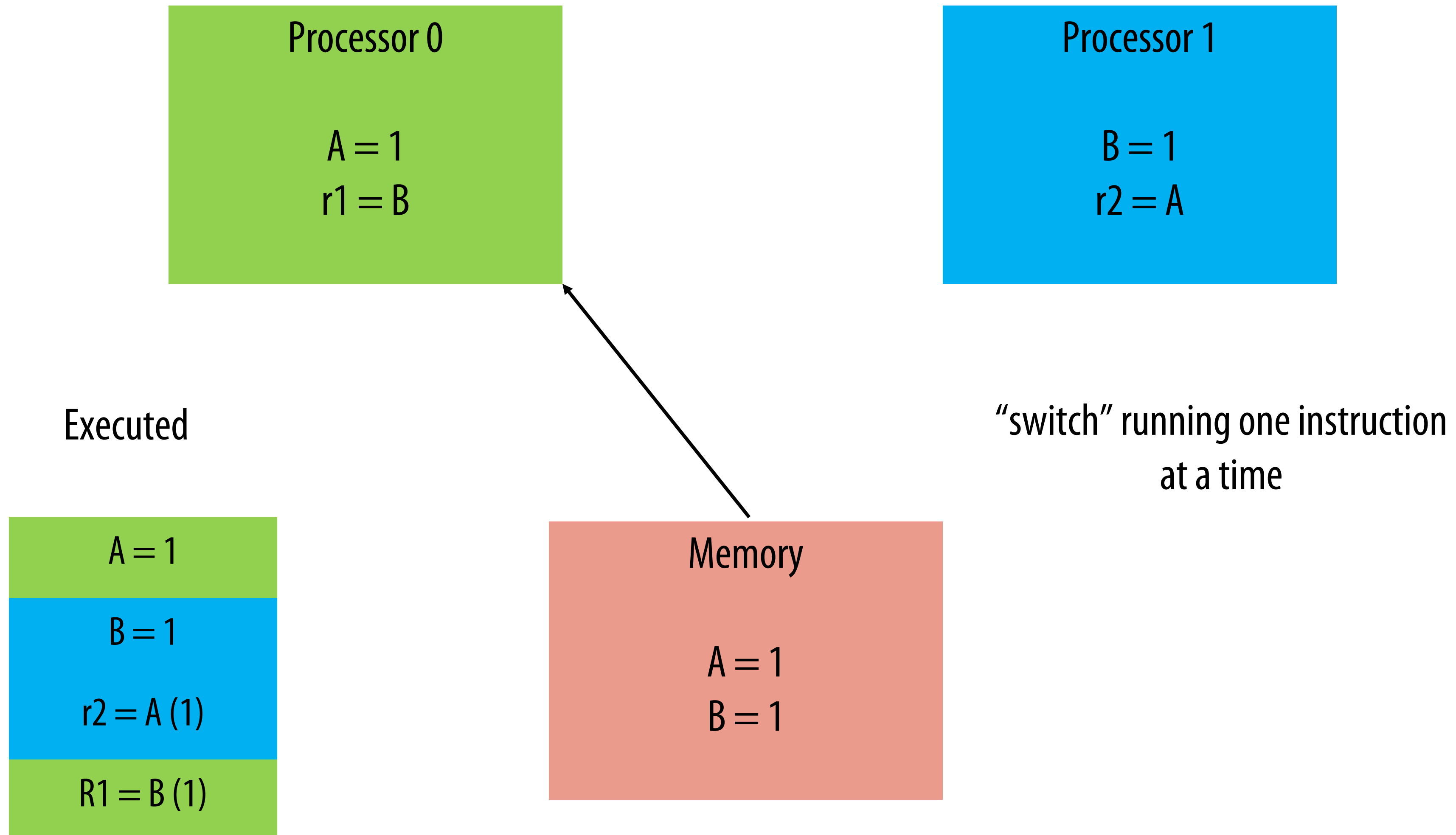
Sequential Consistency Example



Sequential Consistency Example



Sequential Consistency Example

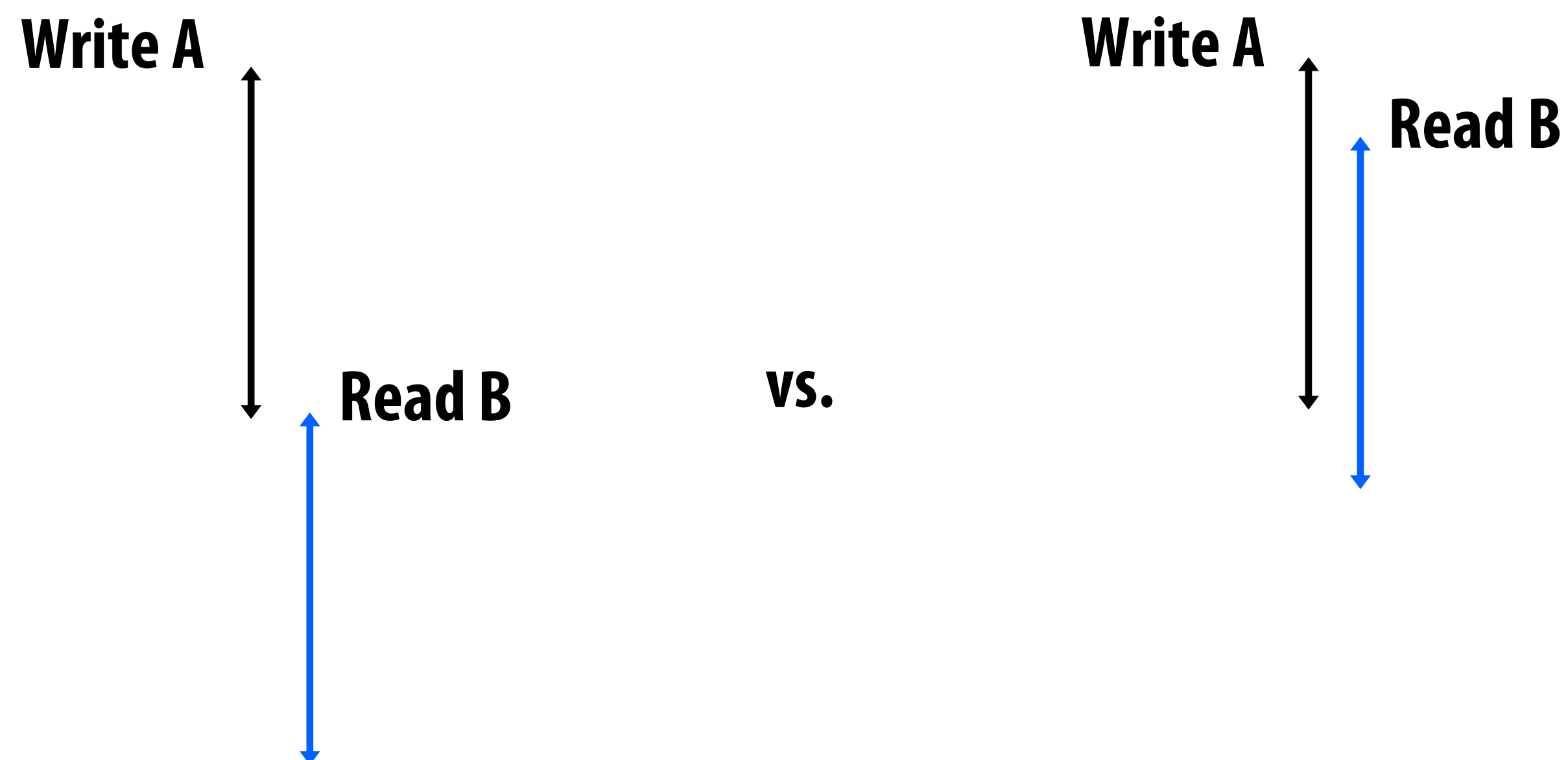


Relaxing memory operation ordering

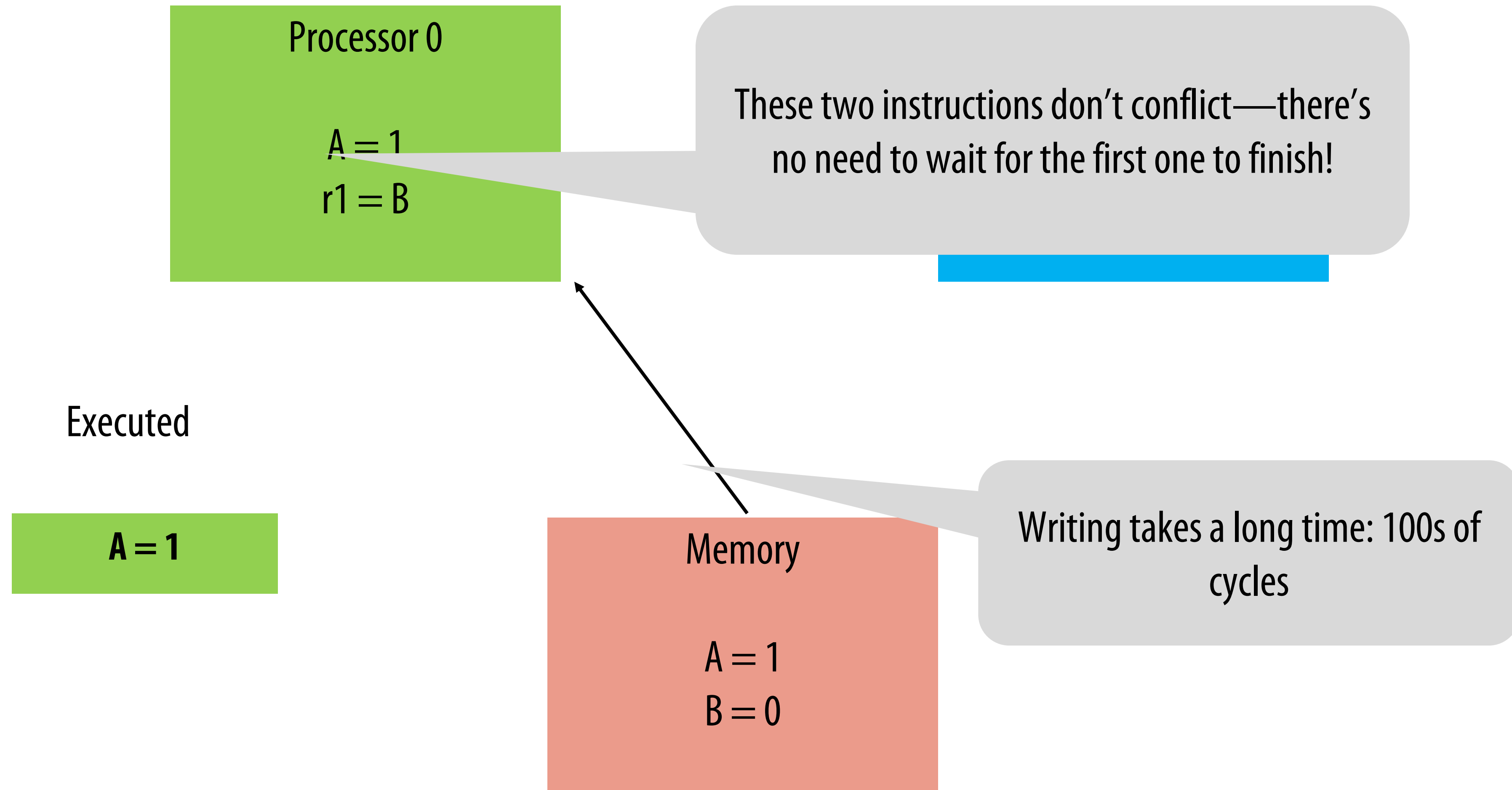
- A sequentially consistent memory system maintains all four memory operation orderings ($W_X \rightarrow R_Y$, $R_X \rightarrow R_Y$, $R_X \rightarrow W_Y$, $W_X \rightarrow W_Y$)
- Relaxed memory consistency models allow certain orderings to be violated

Motivation for relaxed consistency: hiding latency

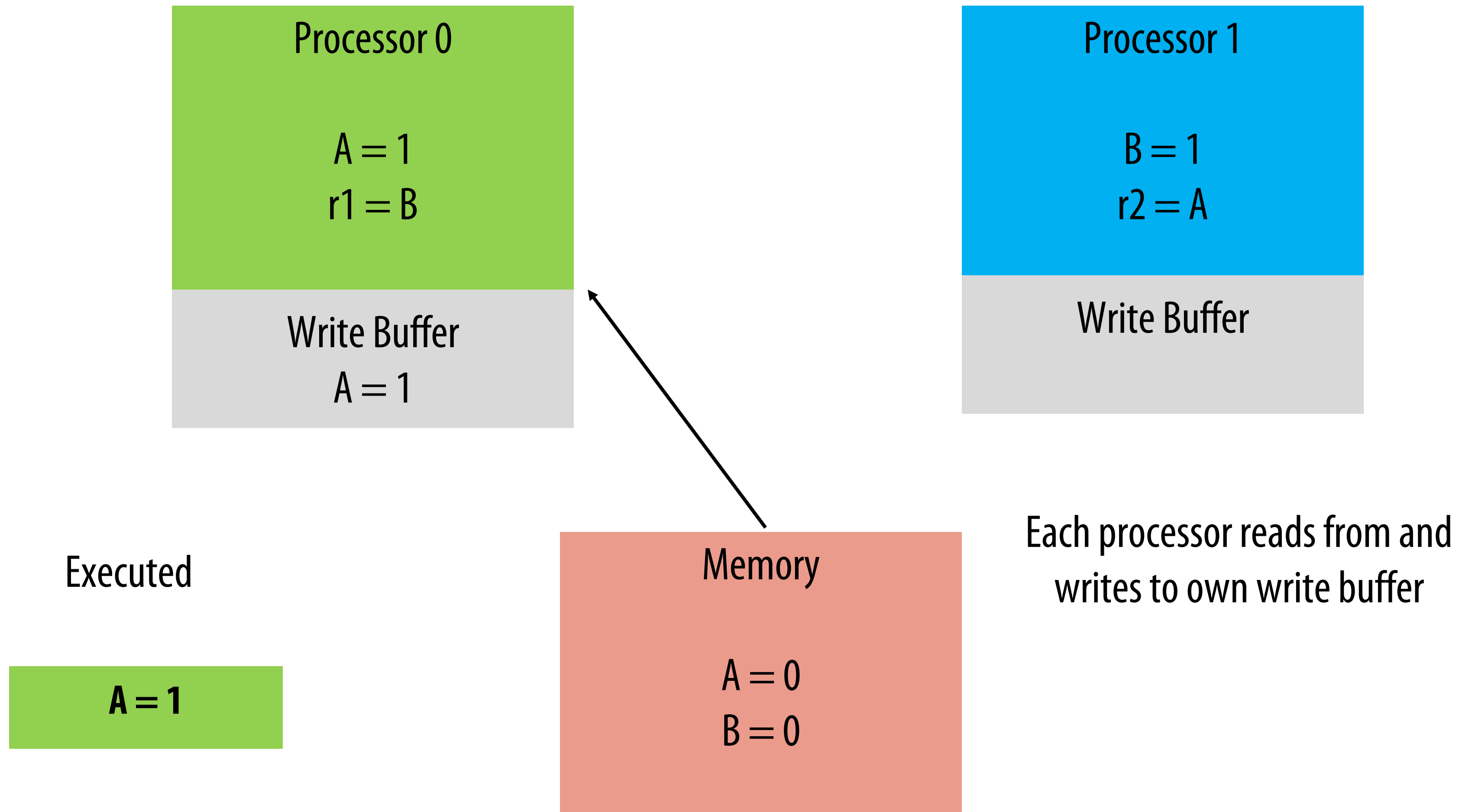
- Why are we interested in relaxing ordering requirements?
 - To gain performance
 - Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
 - Remember, memory access in a cache coherent system may entail much more work than simply reading bits from memory (finding data, sending invalidations, etc.)



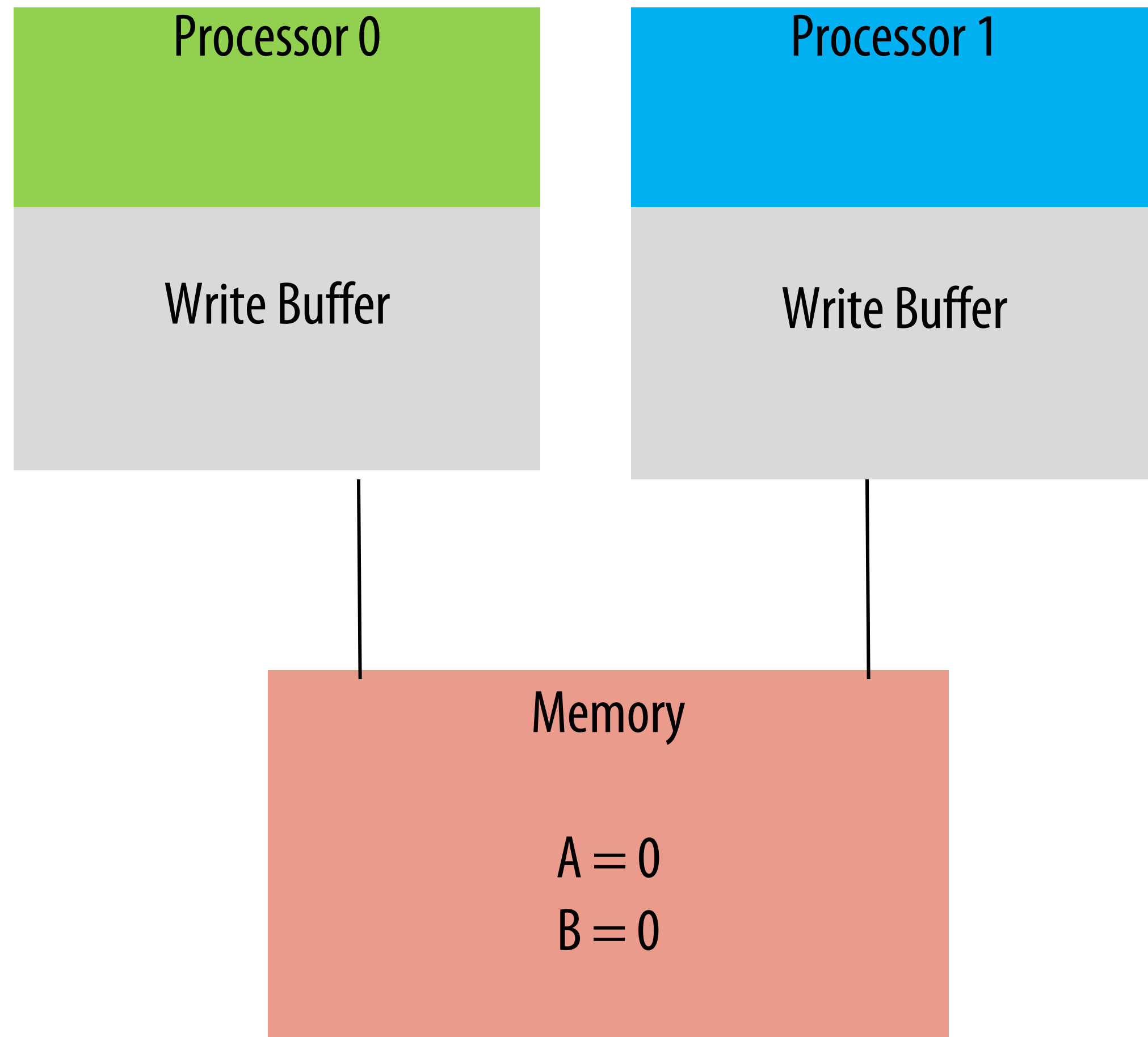
Problem with SC



Optimization: Write Buffer



Write Buffers Change Memory Behavior



Initially $A = B = 0$

Proc 0

(1) $A = 1$

(2) $r1 = B$

Proc 1

(3) $B = 1$

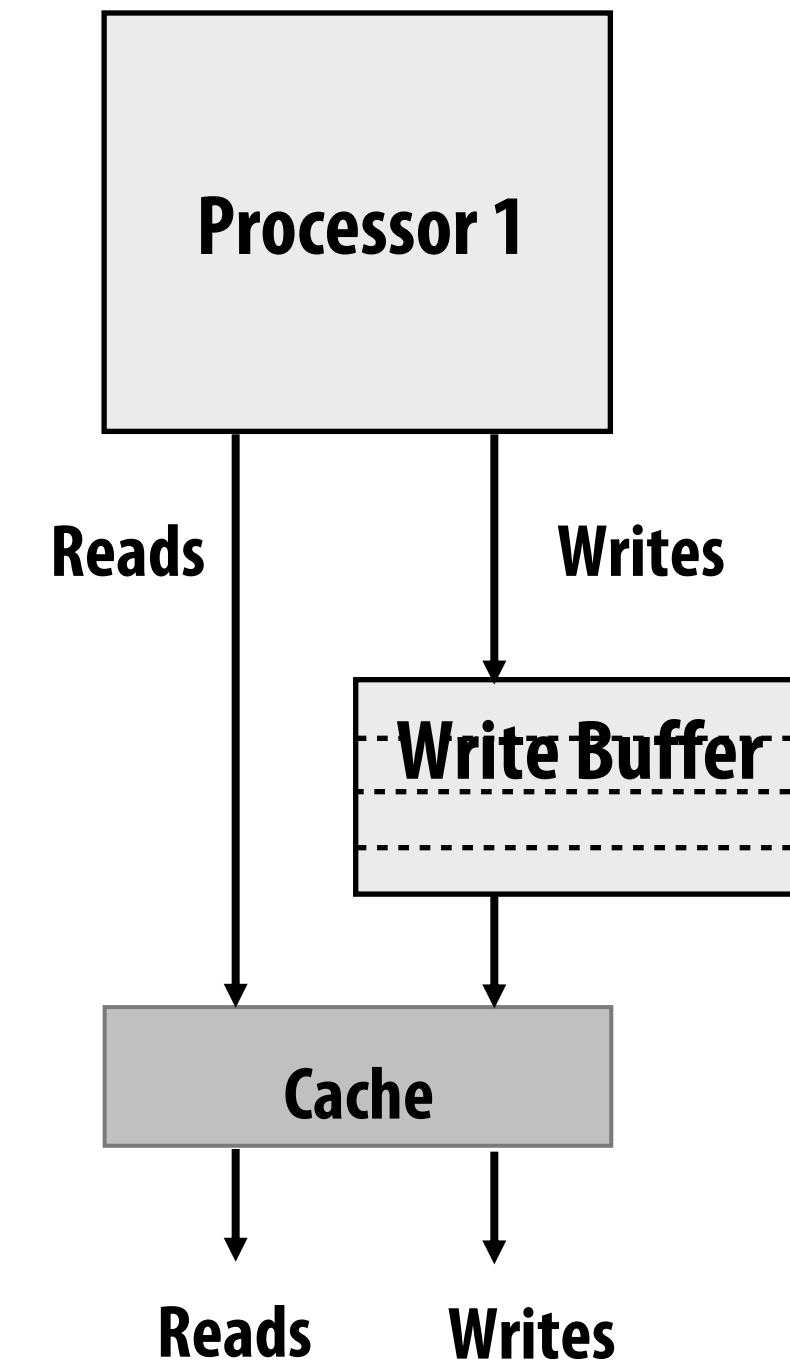
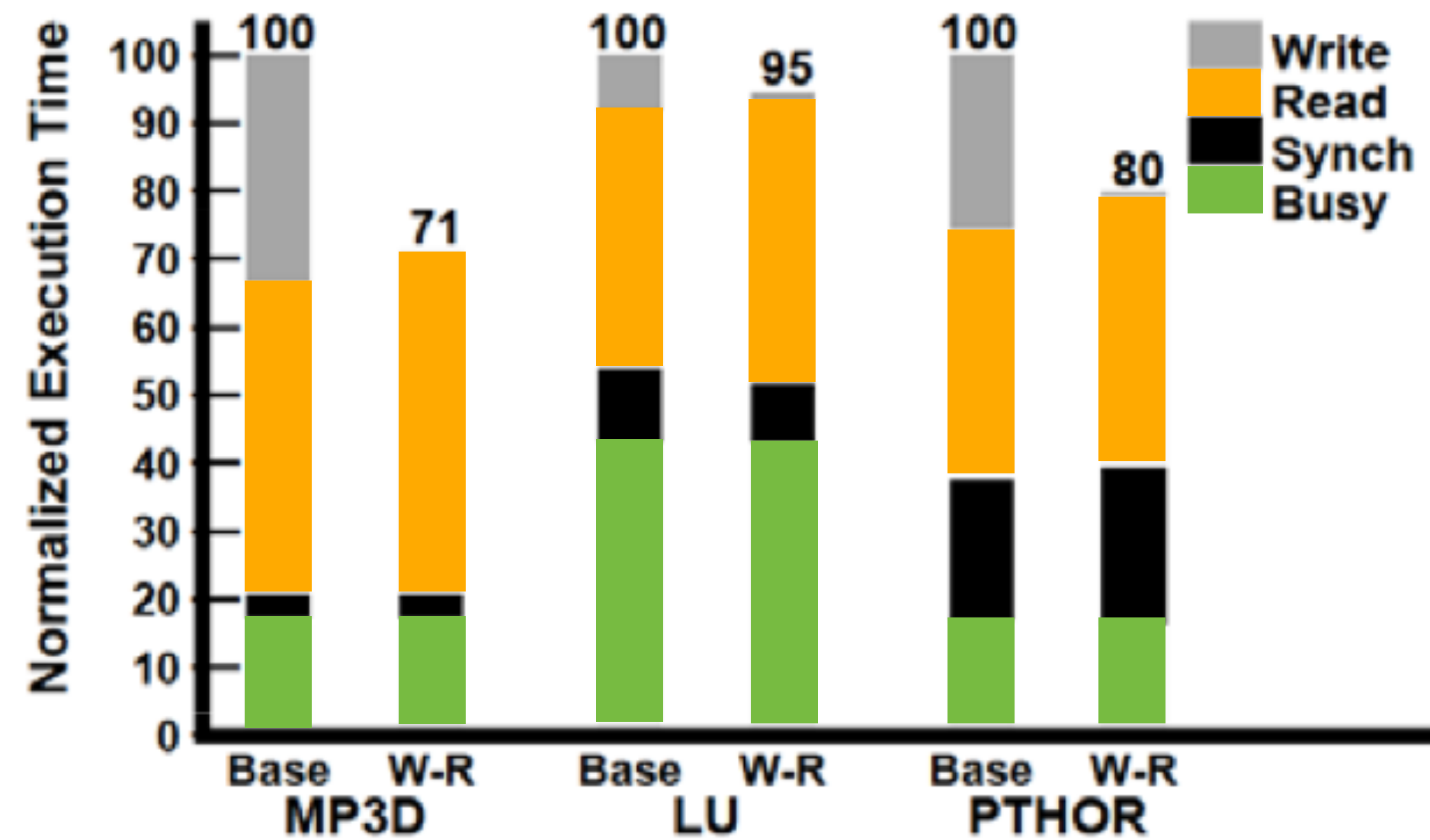
(4) $r2 = A$

Can $r1 = r2 = 0$?

SC: No

Write buffers:

Write buffer performance



Base: Sequentially consistent execution. Processor issues one memory operation at a time, stalls until completion

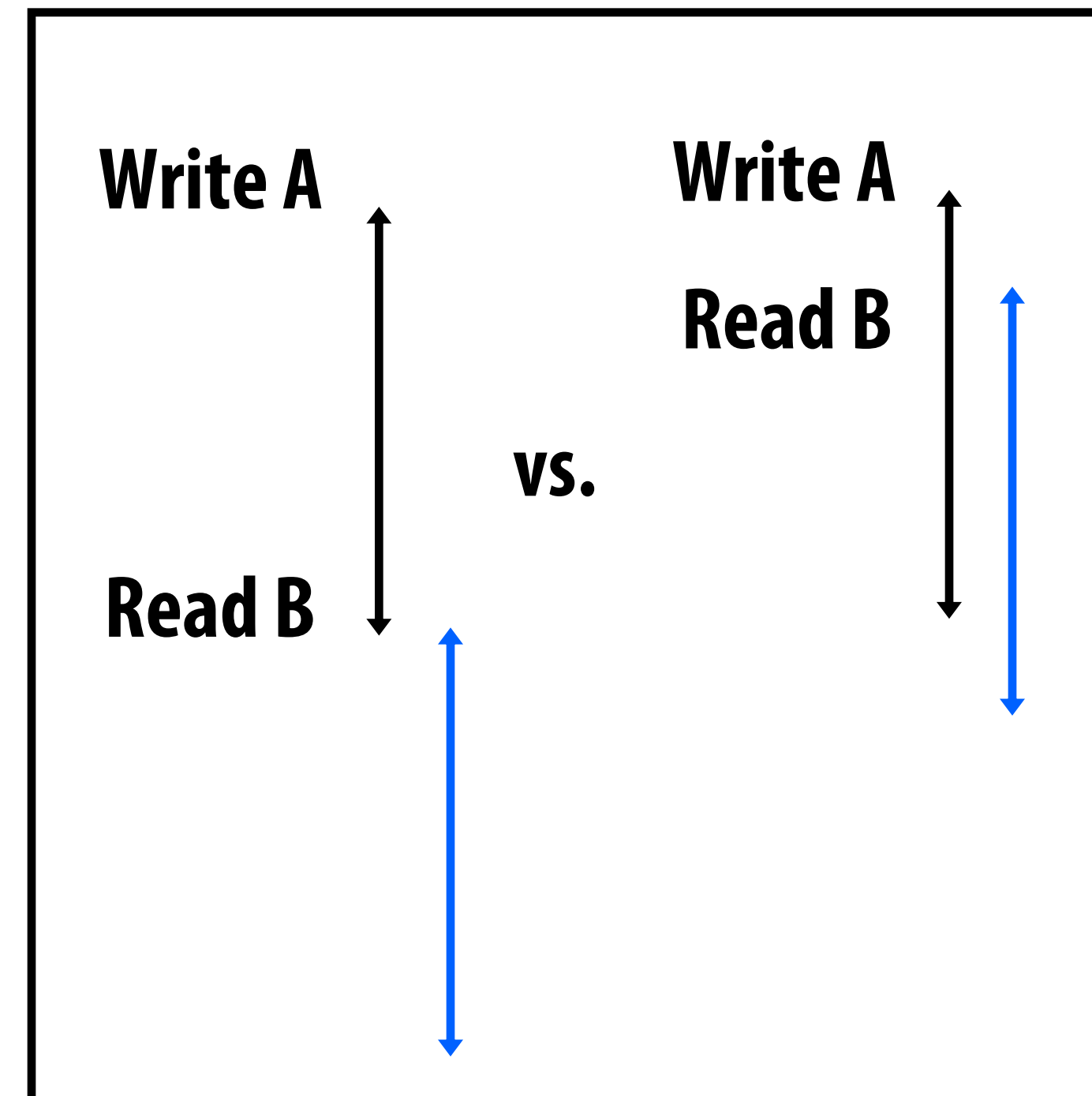
W-R: relaxed $W \rightarrow R$ ordering constraint (write latency almost fully hidden)

Write Buffers: Who Cares?

- Performance improvement
- Every modern processor uses them
 - Intel x86, ARM, SPARC
- Need a weaker memory model
 - TSO: Total Store Order
 - Slightly harder to reason about than SC
 - x86 uses an incompletely specified form of TSO

Allowing reads to move ahead of writes

- Four types of memory operation orderings
 - ~~$W_X \rightarrow R_Y$: write must complete before subsequent read~~
 - $R_X \rightarrow R_Y$: read must complete before subsequent read
 - $R_X \rightarrow W_Y$: read must complete before subsequent write
 - $W_X \rightarrow W_Y$: write must complete before subsequent write
- Allow processor to hide latency of writes
 - Total Store Ordering (TSO)
 - Processor Consistency (PC)



Allowing reads to move ahead of writes

- Total store ordering (TSO)
 - Processor P can read B before its write to A is seen by all processors
(processor can move its own reads in front of its own writes)
 - Reads by other processors cannot return new value of A until the write to A is observed by all processors
- Processor consistency (PC)
 - Any processor can read new value of A before the write is observed by all processors
- In TSO and PC, only $W_x \rightarrow R_y$ order is relaxed. The $W_x \rightarrow W_y$ constraint still exists. Writes by the same thread are not reordered (they occur in program order)

Clarification (make sure you get this!)

- **The cache coherency problem exists because hardware implements the optimization of duplicating data in multiple processor caches. The copies of the data must be kept coherent.**
- **Relaxed memory consistency issues arise from the optimization of reordering memory operations. (Consistency is unrelated to whether or not caches exist in the system.)**

Allowing writes to be reordered

- **Four types of memory operation orderings**

- ~~- $W_X \rightarrow R_Y$: write must complete before subsequent read~~

- $R_X \rightarrow R_Y$: read must complete before subsequent read

- $R_X \rightarrow W_Y$: read must complete before subsequent write

- ~~- $W_X \rightarrow W_Y$: write must complete before subsequent write~~

- **Partial Store Ordering (PSO)**

- **Execution may not match sequential consistency on program 1**

- (P2 may observe change to flag before change to A)**

Thread 1 (on P1)

```
A = 1;
```

```
flag = 1;
```

Thread 2 (on P2)

```
while (flag == 0);
```

```
print A;
```

Why might it be useful to allow more aggressive memory operation reorderings?

- $W \rightarrow W$: processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)
- $R \rightarrow W, R \rightarrow R$: processor might reorder independent instructions in an instruction stream (out-of-order execution)
- Keep in mind these are all valid optimizations if a program consists of a single instruction stream

Allowing all reorderings

- Four types of memory operation orderings
 - ~~- $W_x \rightarrow R_y$: write must complete before subsequent read~~
 - ~~- $R_x \rightarrow R_y$: read must complete before subsequent read~~
 - ~~- $R_x \rightarrow W_y$: read must complete before subsequent write~~
 - ~~- $W_x \rightarrow W_y$: write must complete before subsequent write~~
- No guarantees about operations on data!
 - Everything can be reordered
- Motivation is increased performance
 - Overlap multiple reads and writes in the memory system
 - Execute reads as early as possible and writes as late as possible to hide memory latency
- Examples:
 - Weak ordering (WO)
 - Release Consistency (RC)

Synchronization to the Rescue

- Memory reordering seems like a nightmare (it is!)
- Every architecture provides synchronization primitives to make memory ordering stricter
- Fence (memory barrier) instructions prevent reorderings, but are expensive
 - All memory operations complete before any memory operation after it can begin
- Other synchronization primitives (per address):
 - read-modify-write/compare-and-swap, transactional memory, ...

reorderable reads
and writes here

...

MEMORY FENCE

...

reorderable reads
and writes here

...

MEMORY FENCE

Example: expressing synchronization in relaxed models

- Intel x86/x64 ~ total store ordering
 - Provides sync instructions if software requires a specific instruction ordering not guaranteed by the consistency model
 - `mm_lfence` (“load fence”: wait for all loads to complete)
 - `mm_sfence` (“store fence”: wait for all stores to complete)
 - `mm_mfence` (“mem fence”: wait for all mem operations to complete)
- ARM processors: very relaxed consistency model

A cool post on the role of memory fences in x86:

<http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/>

ARM has some great examples in their programmer’s reference:

[http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier Litmus Tests and Cookbook A08.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf)

A great list:

<http://www.cl.cam.ac.uk/~pes20/weakmemory/>

Problem: Data Races

- Every example so far has involved a data race
 - Two accesses to the same memory location
 - At least one is a write
 - Unordered by synchronization operations

Conflicting data accesses

- Two memory accesses by different processors conflict if...
 - They access the same memory location
 - At least one is a write

- Unsynchronized program
 - Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
 - Unsynchronized programs contain data races: the output of the program depends on relative speed of processors (non-deterministic program results)

Synchronized programs

- Synchronized programs yield SC results on non-SC systems
 - Synchronized programs are data-race-free
- If there are no data races, reordering behavior doesn't matter
 - Accesses are ordered by synchronization, and synchronization forces sequential consistency
- In practice, most programs you encounter will be synchronized (via locks, barriers, etc. implemented in synchronization libraries)
 - Rather than via ad-hoc reads/writes to shared variables like in the example programs

Summary: relaxed consistency

- Motivation: obtain higher performance by allowing reordering of memory operations (reordering is not allowed by sequential consistency)
- One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed
 - But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)
 - Optimize for the common case: most memory accesses are not conflicting, so don't design a system that pays the cost as if they are
- Relaxed consistency models differ in which memory ordering constraints they ignore

Languages Need Memory Models Too

Thread 1

```
X = 0
```

```
for i=0 to 100:
```

```
    X = 1
```

```
    print X
```



Thread 1

```
X = 1
```

```
for i=0 to 100:
```

```
    print X
```

Languages Need Memory Models Too

Optimization not visible to programmer

Thread 1

```
X = 0
for i=0 to 100:
    X = 1
    print X
```

111111111111...

Thread 1

```
X = 1
for i=0 to 100:
    print X
```

111111111111...

Languages Need Memory Models Too

Optimization is visible to programmer

Thread 1

```
X = 0
for i=0 to 100:
  X = 1
  print X
```

111111111111...

11111011111...

Thread 2

```
X = 0
```

Thread 1

```
X = 1
for i=0 to 100:
  print X
```

111111111111...

11111000000...

Thread 2

```
X = 0
```

Provide a contract to programmers about how their memory operations will be reordered by the compiler e.g. no reordering of shared memory operations

Language Level Memory Models

- Modern (C11, C++11) and not-so-modern (Java 5) languages guarantee sequential consistency for data-race-free programs (“SC for DRF”)
 - Compilers will insert the necessary synchronization to cope with the hardware memory model
- No guarantees if your program contains data races!
 - The intuition is that most programmers would consider a racy program to be buggy
- Use a synchronization library!

Memory Consistency Models Summary

- Define the allowed reorderings of memory operations by hardware and compilers
- A contract between hardware or compiler and application software
- Weak models required for good performance?
 - SC can perform well with many more resources
- Details of memory model can be hidden in synchronization library
 - Requires data race free (DRF) programs

Back to Implementing Locks

Test-and-set based lock

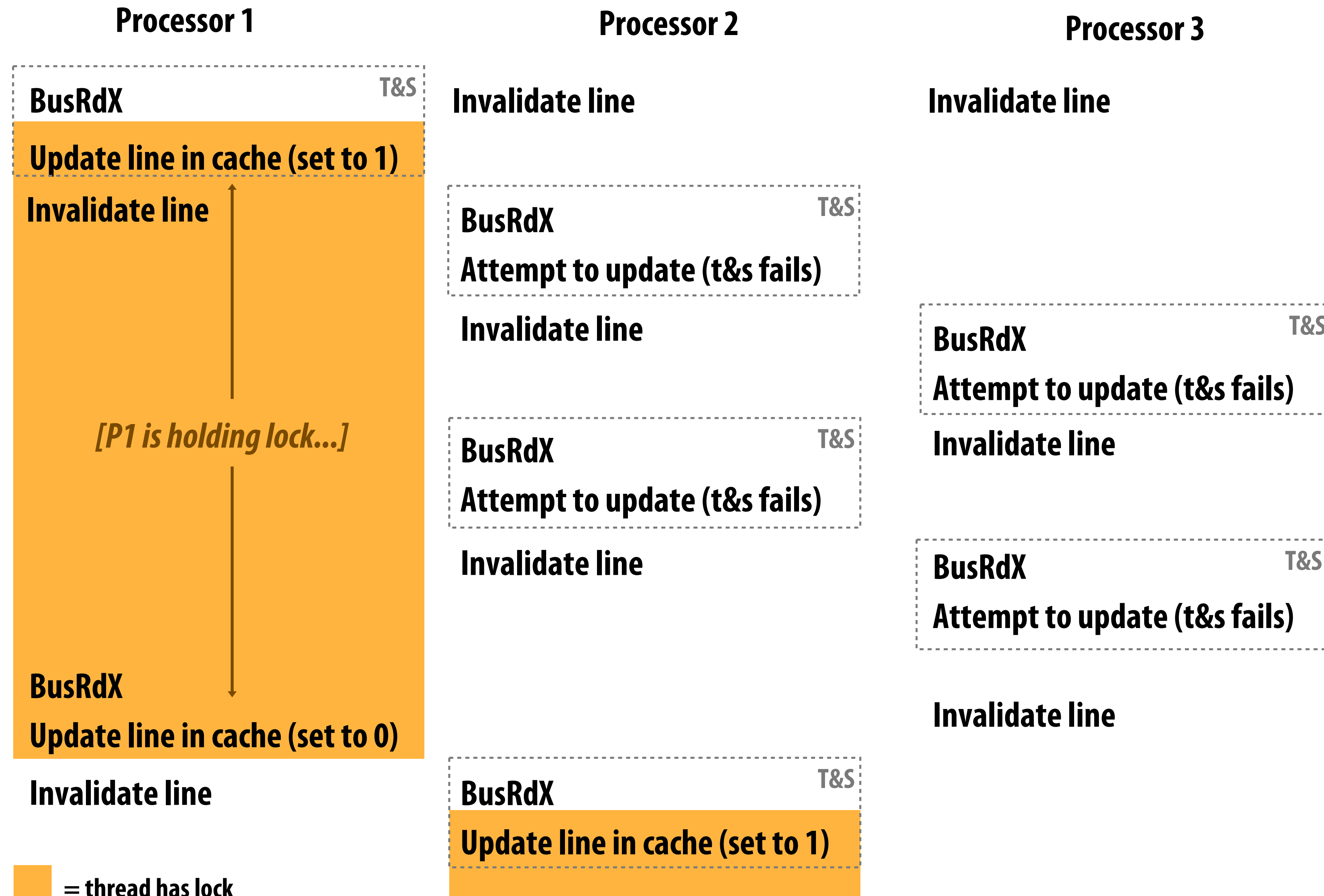
Atomic test-and-set instruction:

```
ts R0, mem[addr]      // load mem[addr] into R0
                       // if mem[addr] is 0, set mem[addr] to 1
```

```
lock:      ts    R0, mem[addr]      // load word into R0
           bnz   R0, lock           // if 0, lock obtained
```

```
unlock:    st    mem[addr], #0      // store 0 to address
```

Test-and-set lock: consider coherence traffic



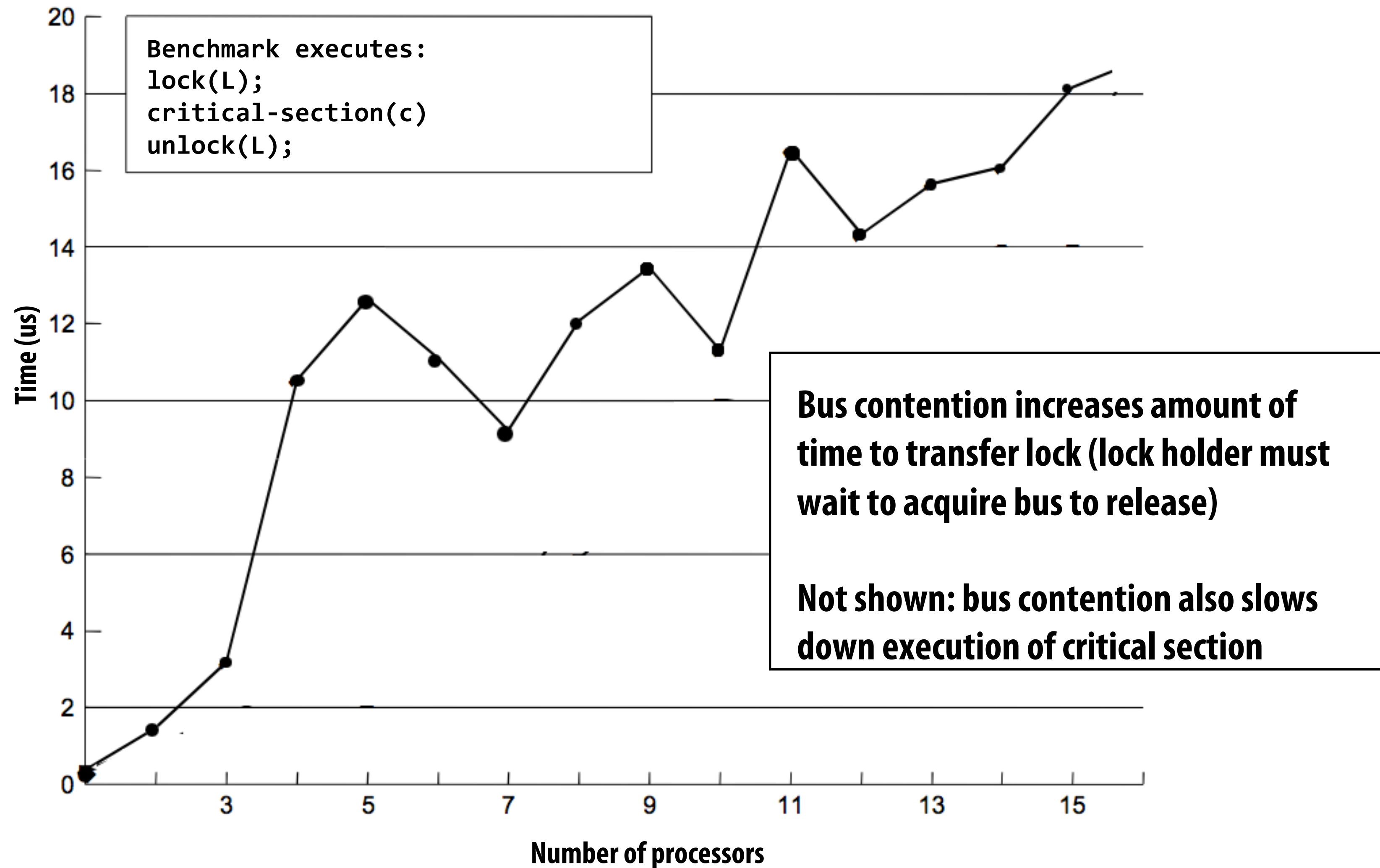
Check your understanding

- **On the previous slide, what is the duration of time the thread running on P1 holds the lock?**
- **At what points in time does P1's cache contain a valid copy of the cache line containing the lock variable?**

Test-and-set lock performance

Benchmark: execute a total of N lock/unlock sequences (in aggregate) by P processors

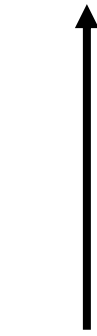
Critical section time removed so graph plots only time acquiring/releasing the lock



x86 cmpxchg

- Compare and exchange (atomic when used with lock prefix)

`lock cmpxchg dst, src`



lock prefix (makes operation atomic)



often a memory address

```
if (dst == EAX)
    ZF = 1
    dst = src
else
    ZF = 0
    EAX = dst
```

x86 accumulator register

flag register

Self-check: Can you implement assembly for atomic compare-and-swap using `cmpxchg`?

```
bool compare_and_swap(int* x, int a, int b) {
    if (*x == a) {
        *x = b;
        return true;
    }

    return false;
}
```


Desirable lock performance characteristics

- **Low latency**
 - If lock is free and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly
- **Low interconnect traffic**
 - If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible
- **Scalability**
 - Latency / traffic should scale reasonably with number of processors
- **Low storage cost**
- **Fairness**
 - Avoid starvation or substantial unfairness
 - One ideal: processors should acquire lock in the order they request access to it

Simple test-and-set lock: low latency (under low contention), high traffic, poor scaling, low storage cost (one int), no provisions for fairness

Test-and-test-and-set lock

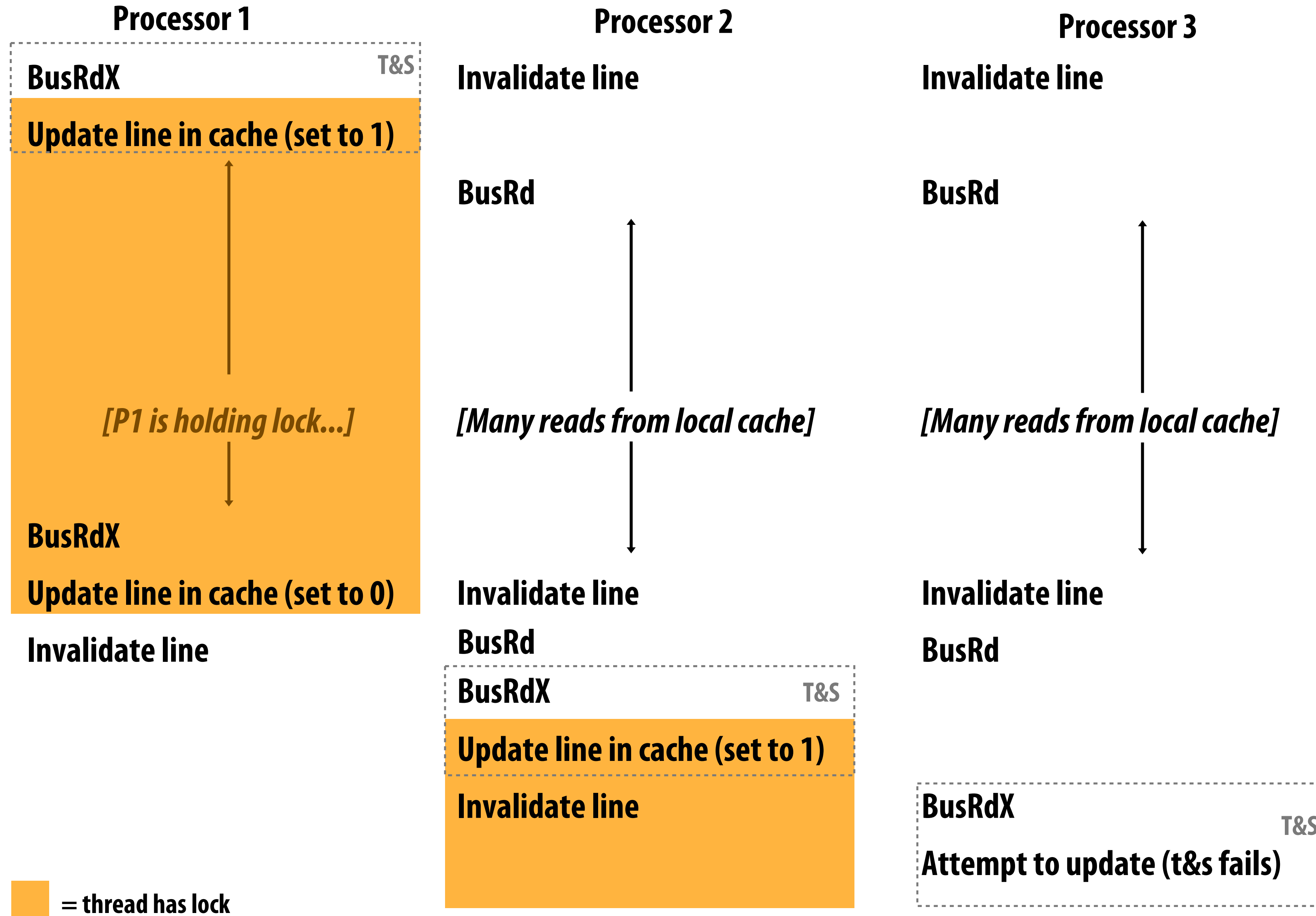
```
void Lock(int* lock) {
    while (1) {

        while (*lock != 0);           // while another processor has the lock...
                                     // (assume *lock is NOT register allocated)

        if (test_and_set(*lock) == 0) // when lock is released, try to acquire it
            return;
    }
}

void Unlock(int* lock) {
    *lock = 0;
}
```

Test-and-test-and-set lock: coherence traffic



Test-and-test-and-set characteristics

- **Slightly higher latency than test-and-set in uncontended case**
 - Must test... then test-and-set
- **Generates much less interconnect traffic**
 - One invalidation, per waiting processor, per lock release ($O(P)$ invalidations)
 - This is $O(P^2)$ interconnect traffic if all processors have the lock cached
 - Recall: test-and-set lock generated one invalidation per waiting processor per test
- **More scalable (due to less traffic)**
- **Storage cost unchanged (one int)**
- **Still no provisions for fairness**

Additional atomic operations

Atomic operations provided by CUDA

```
int    atomicAdd(int* address, int val);
float  atomicAdd(float* address, float val);
int    atomicSub(int* address, int val);
int    atomicExch(int* address, int val);
float  atomicExch(float* address, float val);
int    atomicMin(int* address, int val);
int    atomicMax(int* address, int val);
unsigned int atomicInc(unsigned int* address, unsigned int val);
unsigned int atomicDec(unsigned int* address, unsigned int val);
int    atomicCAS(int* address, int compare, int val);
int    atomicAnd(int* address, int val); // bitwise
int    atomicOr(int* address, int val);  // bitwise
int    atomicXor(int* address, int val); // bitwise
```

(omitting additional 64 bit and unsigned int versions)

Implementing atomic fetch-and-op

```
// atomicCAS:  
// atomic compare and swap performs the following logic atomically  
int atomicCAS(int* addr, int compare, int new) {  
    int old = *addr;  
    *addr = (old == compare) ? new : old;  
    return old;  
}
```

Exercise: how can you build an atomic fetch+op out of atomicCAS()?

Example: atomic_min()

```
int atomic_min(int* addr, int x) {  
    int old = *addr;  
    int new = min(old, x);  
    while (atomicCAS(addr, old, new) != old) {  
        old = *addr;  
        new = min(old, x);  
    }  
}
```

What about these operations?

```
int atomic_increment(int* addr, int x); // for signed values of x  
void lock(int* addr);
```

Load-linked, Store Conditional (LL/SC)

- **Pair of corresponding instructions (not a single atomic instruction like compare-and-swap)**
 - **load_linked(x): load value from address**
 - **store_conditional(x, value): store value to x, if x hasn't been written to since corresponding LL**
- **Corresponding ARM instructions: LDREX and STREX**
- **How might LL/SC be implemented on a cache coherent processor?**

Simple Spin Lock with LL/SC

```
lock: ll reg1, lockvar      /* LL lockvar to reg1 */
      sc lockvar, reg2      /* SC reg2 into lockvar */
      beqz reg2, lock       /* if false, start again */
      bnzreg1, lock        /* if locked, start again */
      ret
```

```
unlock: st location, #0    /* write 0 to location */
      ret
```

- Can do more fancy atomic ops by changing what's between LL & SC
 - But keep it small so SC likely to succeed
 - Don't include instructions that would need to be undone (e.g. stores)
- LL/SC are not lock, unlock respectively
 - Only guarantee no conflicting write to lock variable between them
 - But can use directly to implement simple operations on shared variables

Loop Parallelism (LLP)

- **Overwhelming majority of scientific/engineering applications are expressed in terms of iterative constructs, that is, loops**
 - **Focus on parallelizing loops**
- **Particular useful approach if starting from an existing program**
 - **Major restructuring is impractical/unnecessary**
- **Goal of exploiting LLP is to evolve the sequential program into a parallel program**
 - **Through transformations that leave the program semantics unchanged**
- **LLP works well for shared address space (e.g. Multicore)**

Parallel Loops

- ```
for (i = 0; i < n; i++) {
 A[i] = A[i] + B;
}
```
  
- ```
for (i = 1; i < n; i++) {  
    A[i] = A[i-1] + C[i-1];    /* S1 */  
    B[i] = B[i-1] + A[i];     /* S2 */  
}
```

Parallel Loops

- ```
for (i = 0; i < n; i++) {
 A[i] = A[i] + B[i]; /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

# Data Parallelism with OpenMP

For-loop with independent iterations

```
for (i = 0; i < n; i++)
 c[i] = a[i] + b[i];
```

For-loop parallelized using  
an OpenMP pragma

```
#pragma omp parallel for \
 shared(n, a, b, c) \
 private(i)
for (i = 0; i < n; i++)
 c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```

```
gcc source.c -fopenmp
```

# Privatizing Variables

- **Critical to performance!**
- **OpenMP pragmas:**
  - **Designed to make parallelizing sequential code easier**
  - **Makes copies of “private” variables *automatically***
    - **And performs some automatic initialization, too**
  - **Must specify shared/private per-variable in parallel region**
    - **private: Uninitialized private data**
      - **Private variables are undefined on entry and exit of the parallel region**
    - **shared: All-shared data**
    - **threadprivate: “static” private for use across several parallel regions**

# Firstprivate/Lastprivate Clauses

- `firstprivate (list)`

- All variables in the list are initialized with the value the original object had before entering the parallel region

64

- `lastprivate (list)`

- The thread that executes the last iteration or section in sequential order updates the value of the objects in the list

# Example Private Variables

```
main()
{
 A = 10;

 #pragma omp parallel
 {
 #pragma omp for private(i) firstprivate(A) lastprivate(B)...
 for (i=0; i<n; i++)
 {

 B = A + i; /*-- A undefined, unless declared
 firstprivate --*/

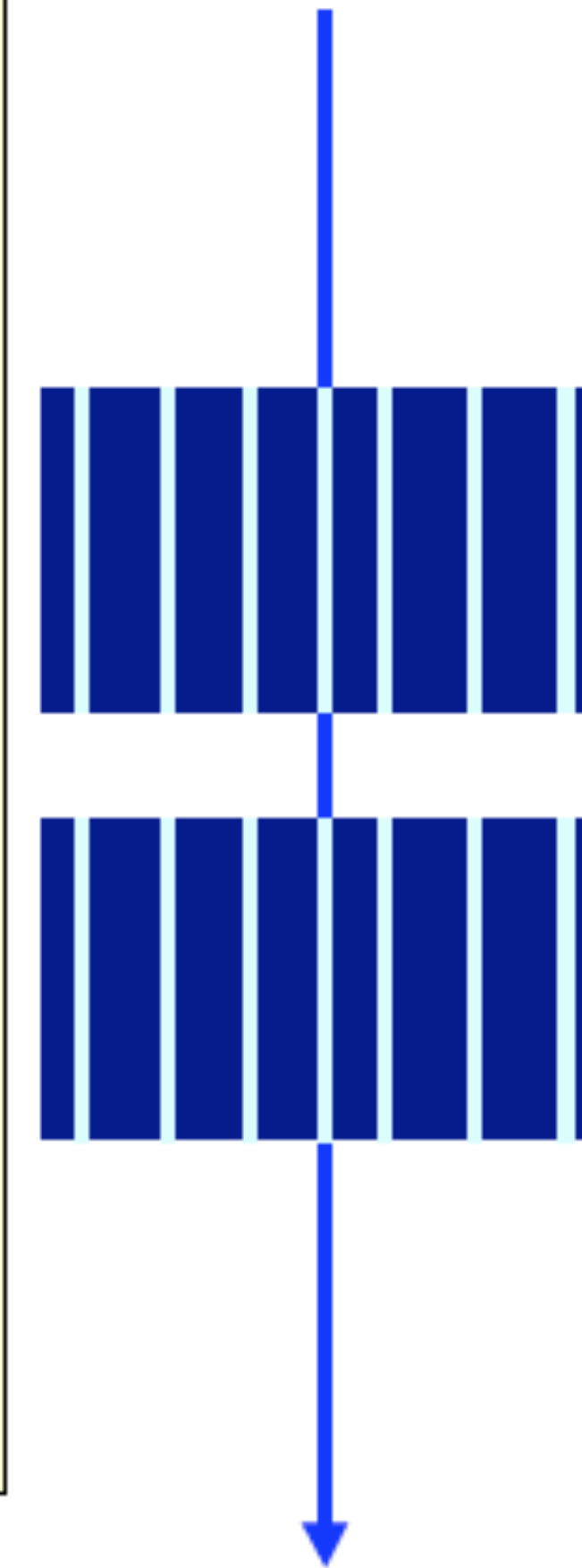
 }

 C = B; /*-- B undefined, unless declared
 lastprivate --*/
 } /*-- End of OpenMP parallel region --*/
}
```



# for directive Example

```
#pragma omp parallel default(none) \
 shared(n,a,b,c,d) private(i)
{
 #pragma omp for
 for (i=0; i<n-1; i++)
 b[i] = (a[i] + a[i+1])/2;
 #pragma omp for
 for (i=0; i<n; i++)
 d[i] = 1.0/c[i];
} /*-- End of parallel region --*/
 (implied barrier)
```



# Nested Loop Parallelism

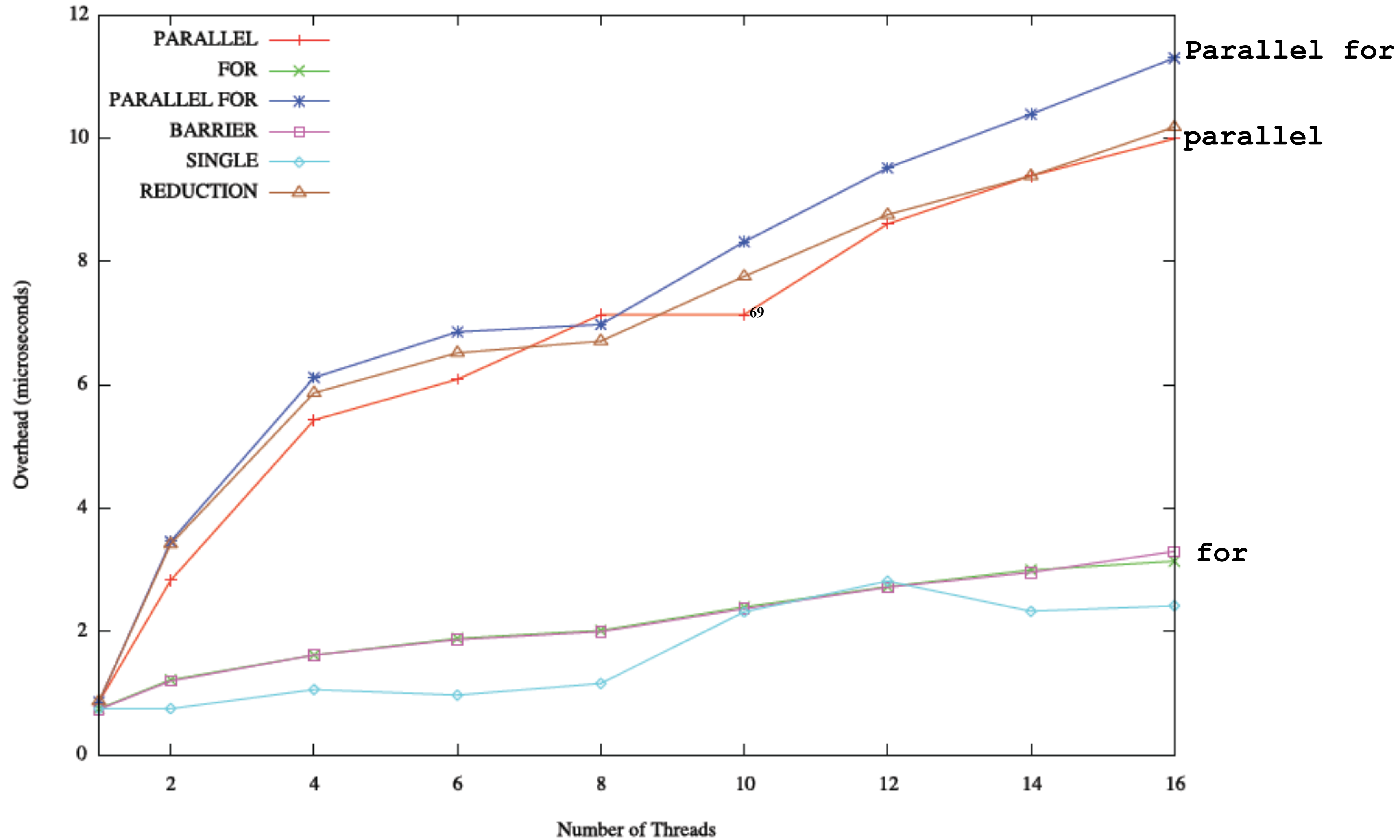
```
#pragma omp parallel for
 for(int y=0; y<25; ++y)
 {
 #pragma omp parallel for
 for(int x=0; x<80; ++x)
 tick(x,y);
 }
```

# Multiple Part Parallel Regions

- You can also have a “multi-part” parallel region
  - Allows easy alternation of serial & parallel parts
  - Doesn't require re-specifying # of threads, etc.

```
#pragma omp parallel . . .
{
 #pragma omp for
 . . . Loop here . . .
 #pragma omp single
 . . . Serial portion here . . .
 #pragma omp sections
 . . . Sections here . . .
}
```

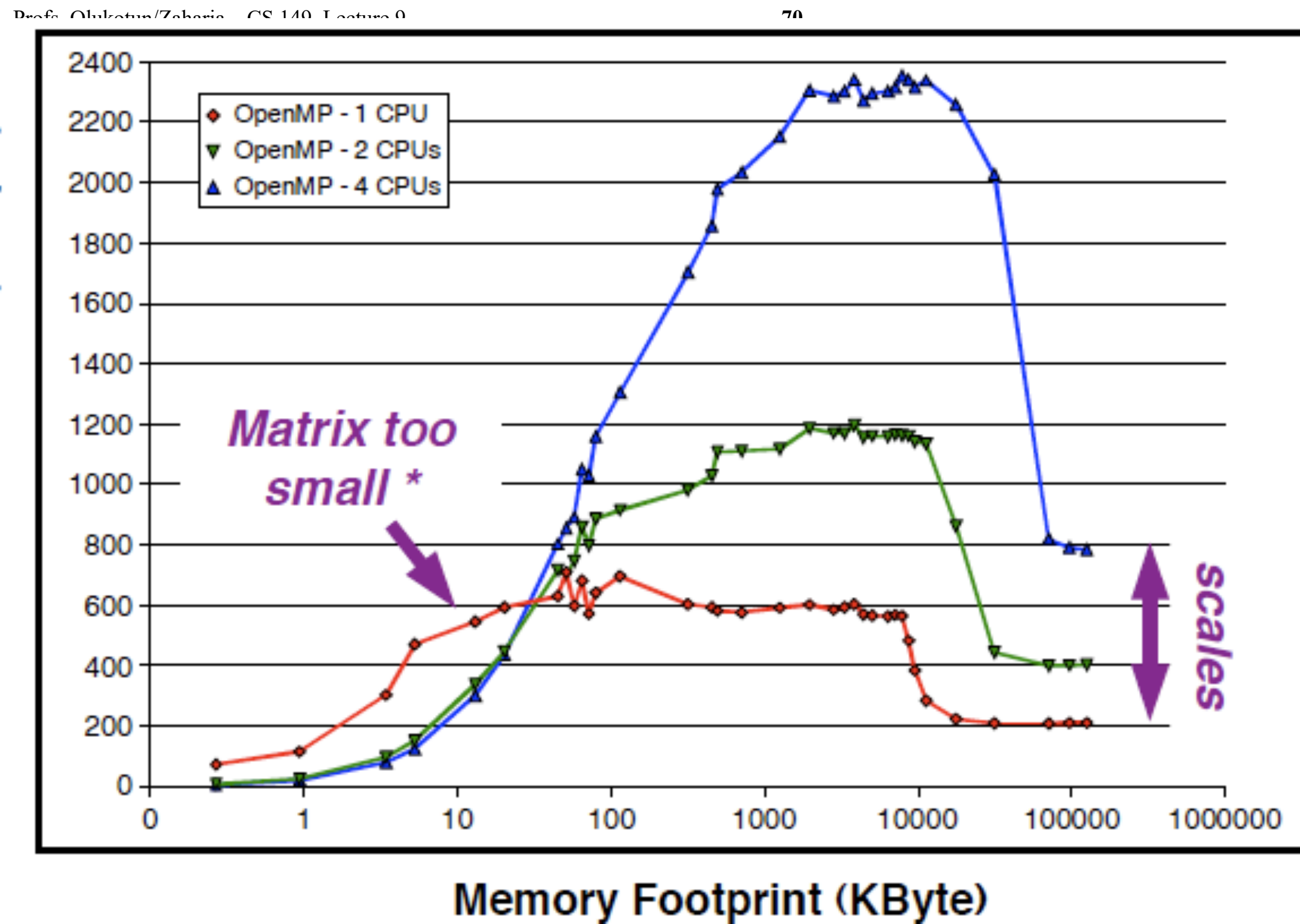
# OMP Directives Overheads



# “if” Clause

- if (scalar expression)
  - Only execute in parallel if expression evaluates to true
  - Otherwise, execute serially

```
#pragma omp parallel if (n > threshold) \
 shared(n,x,y) private(i)
{
 #pragma omp for
 for (i=0; i<n; i++)
 x[i] += y[i];
} /*-- End of parallel region --*/
```



Performance without if clause

# Reductions in OpenMP

- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

Prof. Olukotun/Zaharia CS 149 Lecture 9

71

- The reduction clause has this syntax:  
`reduction (<op> :<variable>)`

- Operators

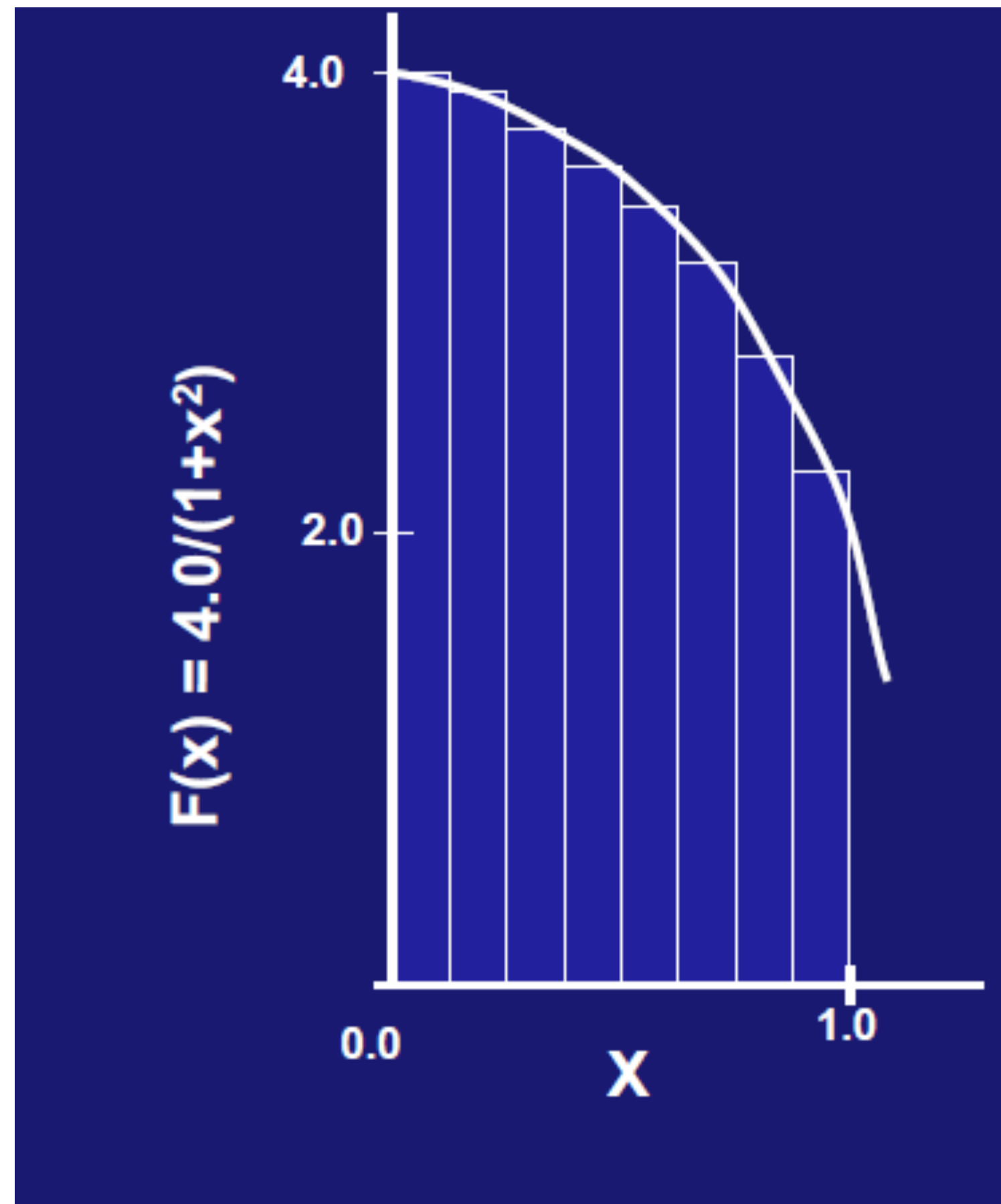
– +            Sum

– \*            Product

– &, |, ^      Bitwise and, or, exclusive or

– &&, ||      Logical and, or

# Example: Numerical Integration



- We know mathematically that

$$\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx$$

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

- We can approximate the integral as a sum of rectangles:

# Sequential Pi Computation

```
static long num_steps = 100000;
double step;

void main () {
 int i; double x, pi, sum = 0.0;
 step = 1.0/(double) num_steps;
 for (i=0;i< num_steps; i++){
 x = (i+0.5)*step;
 sum = sum + 4.0/(1.0+x*x);
 }
 pi = step * sum;
}
```



# Loop Parallelized Pi Computation

```
#include <omp.h>
static long num_steps = 1000000; double step;
#define NUM_THREADS 8

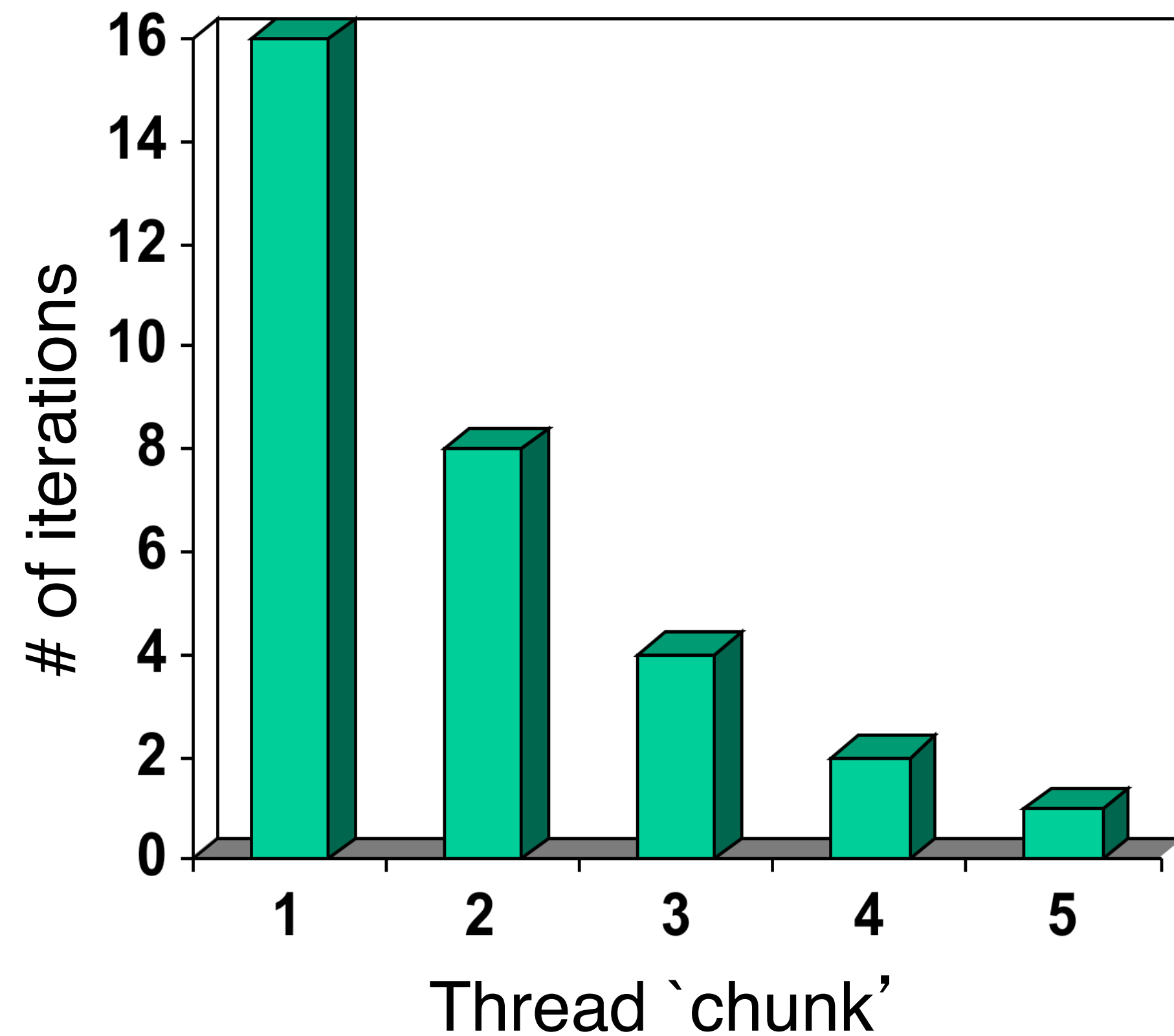
void main (){
 int i; double x, pi, sum = 0.0;
 step = 1.0/(double) num_steps;
 omp_set_num_threads(NUM_THREADS);
 #pragma omp parallel for private(x) reduction(+:sum)
 for (i=0;i< num_steps; i++){
 x = (i+0.5)*step;
 sum = sum + 4.0/(1.0+x*x);
 }
 pi = step * sum;
}
```

- Notice that we haven't changed any lines of code, only added 4 lines
- Compare to MPI

# Dynamic Tasking with OpenMP

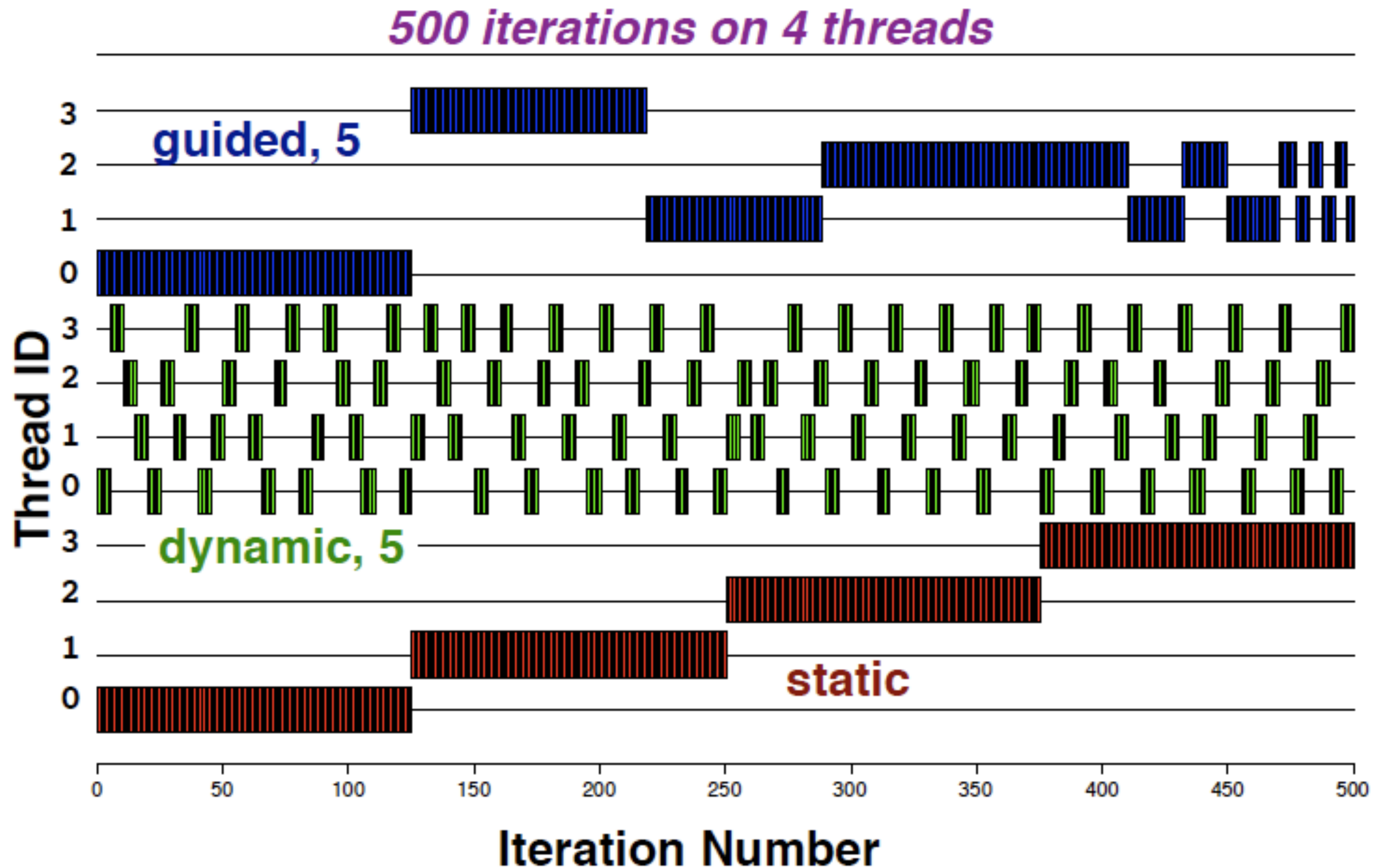
- **OpenMP is a mixed bag**
  - **schedule(dynamic, size) is a dynamic equivalent to the static directive**
    - **Master passes off values of iterations to the workers of size size**
    - **Automatically handles dynamic tasking of simple loops**
  - **Otherwise must make your own**
    - **Includes many commonly used cases, unlike static**
    - **Just like pthreads, except must be lock-only**

# OpenMP Guided Scheduling



- `schedule(guided, size)`
- Guided scheduling is a compromise to reduce scheduling overhead
- Iteration space is divided up into exponentially decreasing chunks
- Final size is usually 1, unless set by the programmer
- Chunks of work are dynamically obtained
- Works quite well provided work per iteration is constant – if unknown dynamic is better

# OpenMP Scheduling



# Tasking in OpenMP 3.0

- **Tasking allows parallelization of units of work that are dynamically generated**
- **Provides flexible model for irregular parallelism**
- **#pragma omp task [clause [,]clause] ...]**
  - **structured-block**
- **Task Synchronization**
  - **C/C++: #pragma omp taskwait**
  - **Current task suspends execution until all children tasks, generated within the current task up to this point, are complete**

# Fibonacci Example

- Default for local variables is firstprivate

```
int fib (int n)
{
 int x,y;
 if (n < 2) return n;
#pragma omp task shared(x)
 x = fib(n-1);
#pragma omp task shared(y)
 y = fib(n-2);
#pragma omp taskwait
 return x+y;;
}
```

# OpenMP Summary

- **OpenMP provides a simple programming model**
  - **Loops or sections**
  - **Incremental parallelism**
- **Targeted at shared memory systems**
  - **Won't scale easily to large machines**
  - **Easy to create false sharing**
- **Compilers with OpenMP 2.5 support are widely available**
- **OpenMP 3.0 supports tasking**
  - **Supports irregular parallelism**