Chapter 2: Machine Models

Elements of Parallel Computing

Eric Aubanel

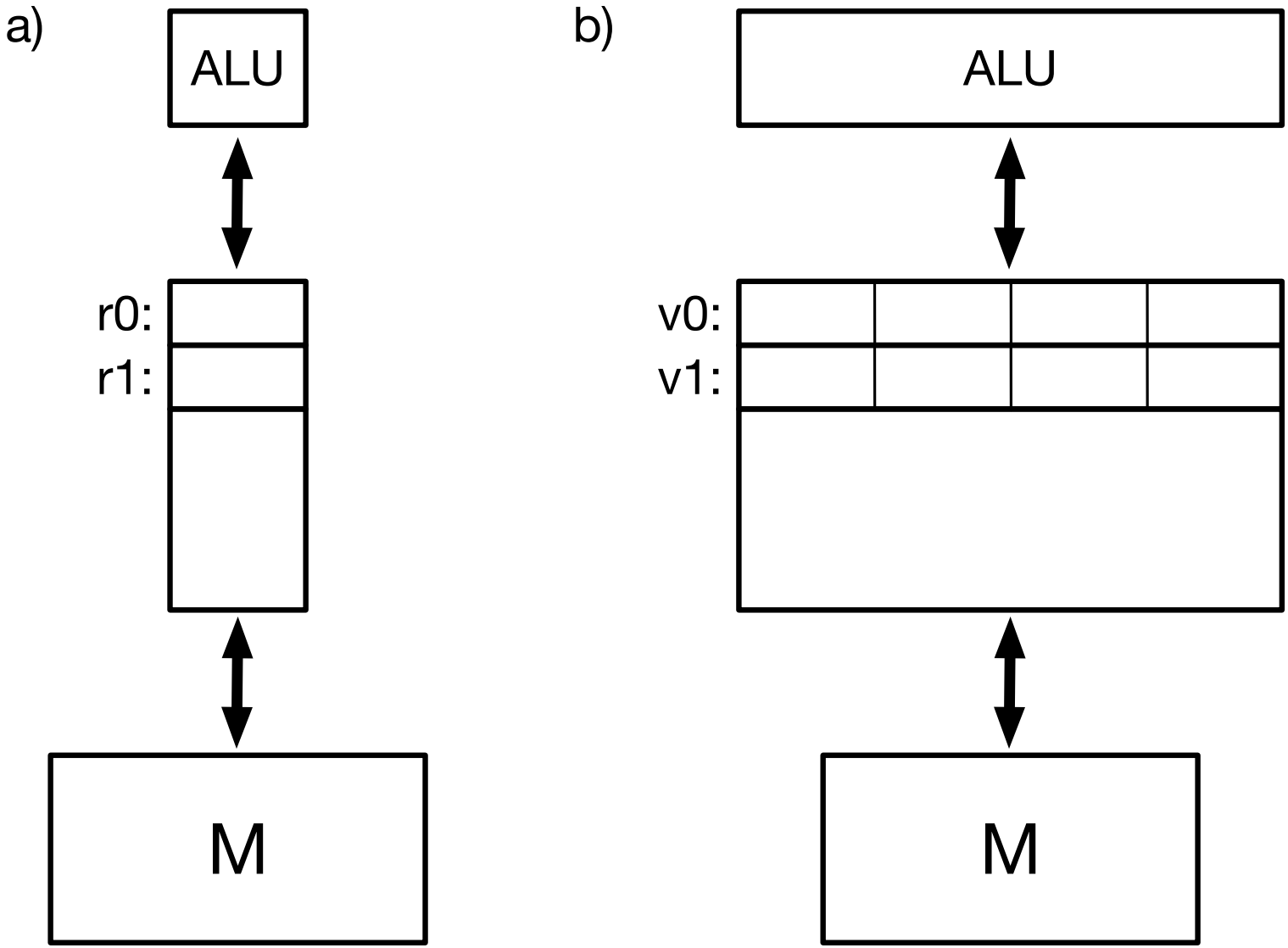
Outline

Three machine models:

- SIMD
- Shared memory
  - Multicore: **latency-oriented**
  - Manycore: **Throughput-oriented** (*Understanding Throughput-Oriented Architectures*, Garland and Kirk, CACM, Nov. 2010)
- Distributed memory

More than one model can be present in one computer

SIMD



Scalar ALU: $c \leftarrow a + b$

1: $r1 \leftarrow$ load $a[0]$
2: $r2 \leftarrow$ load $b[0]$
3: $r2 \leftarrow$ add $r1, r2$
4: $c[0] \leftarrow$ store $r2$
5: $r1 \leftarrow$ load $a[1]$
6: $r2 \leftarrow$ load $b[1]$
7: $r2 \leftarrow$ add $r1, r2$
8: $c[1] \leftarrow$ store $r2$
9: $r1 \leftarrow$ load $a[2]$
10: $r2 \leftarrow$ load $b[2]$
11: $r2 \leftarrow$ add $r1, r2$
12: $c[2] \leftarrow$ store $r2$
13: $r1 \leftarrow$ load $a[3]$
14: $r2 \leftarrow$ load $b[3]$
15: $r2 \leftarrow$ add $r1, r2$
16: $c[3] \leftarrow$ store $r2$

SIMD ALU: $c \leftarrow a + b$

1: $v1 \leftarrow$ vload a
2: $v2 \leftarrow$ vload b
3: $v2 \leftarrow$ vadd $v1, v2$
4: $c \leftarrow$ vstore $v2$

SIM**T** $c \leftarrow a + b$

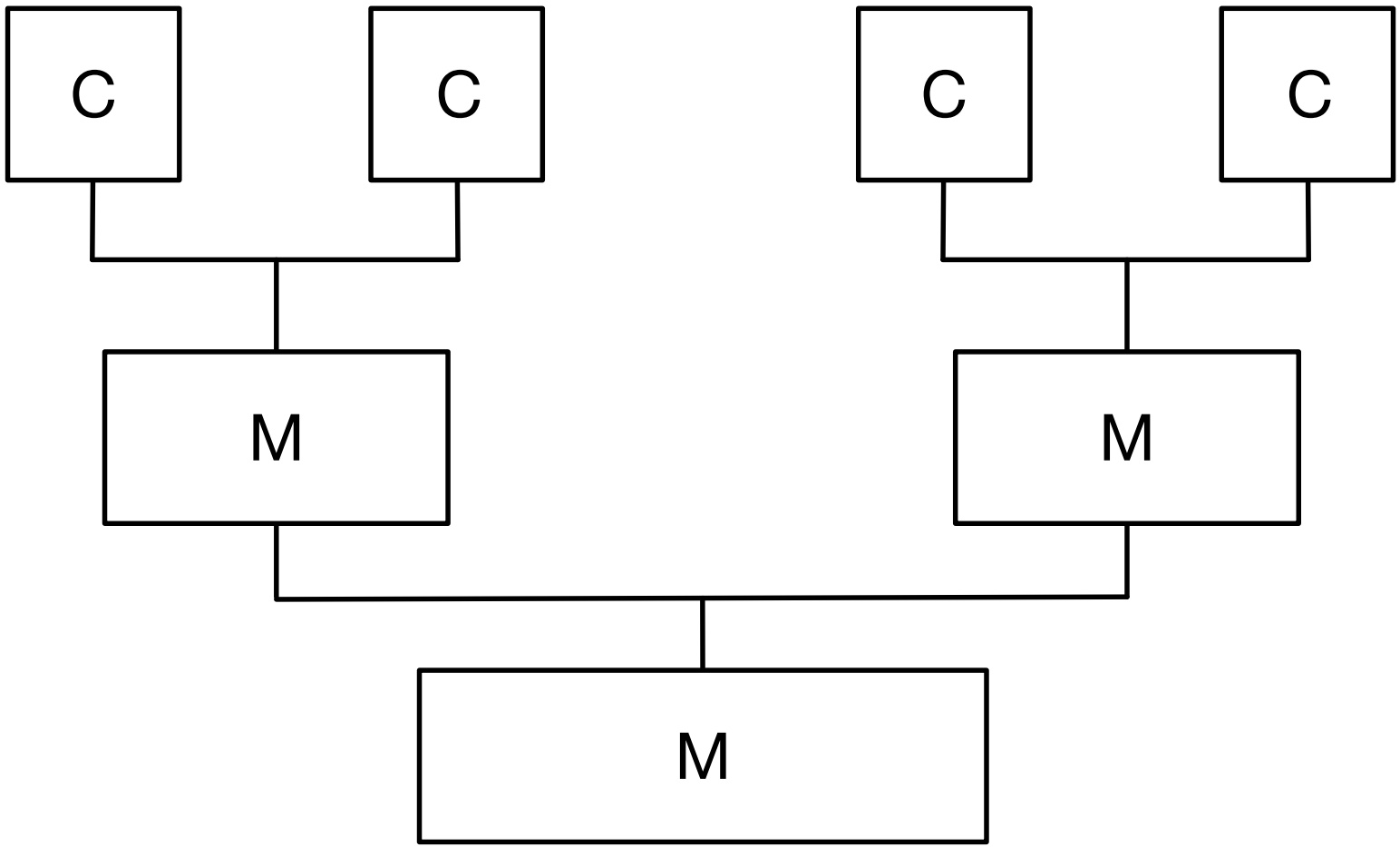
Nvidia GPUs: Single Instruction Multiple **Threads**: group (*warp*) of threads executes in SIMD fashion

t0:	$r00 \leftarrow$ load $a[0]$	$r10 \leftarrow$ load $a[1]$	$\cdots$
t1:	$r01 \leftarrow$ load $b[0]$	$r11 \leftarrow$ load $b[1]$	$\cdots$
t2:	$r01 \leftarrow$ add $r00, r01$	$r11 \leftarrow$ add $r10, r11$	$\cdots$
t3:	$c[0] \leftarrow$ store $r01$	$c[1] \leftarrow$ store $r11$	$\cdots$

SIMT Model

- Programmer writes code for single thread
  - SPMD programming model (see later)
- Warps of threads execute in SIMD fashion
  - Not the same type of threads as in multicore processsor

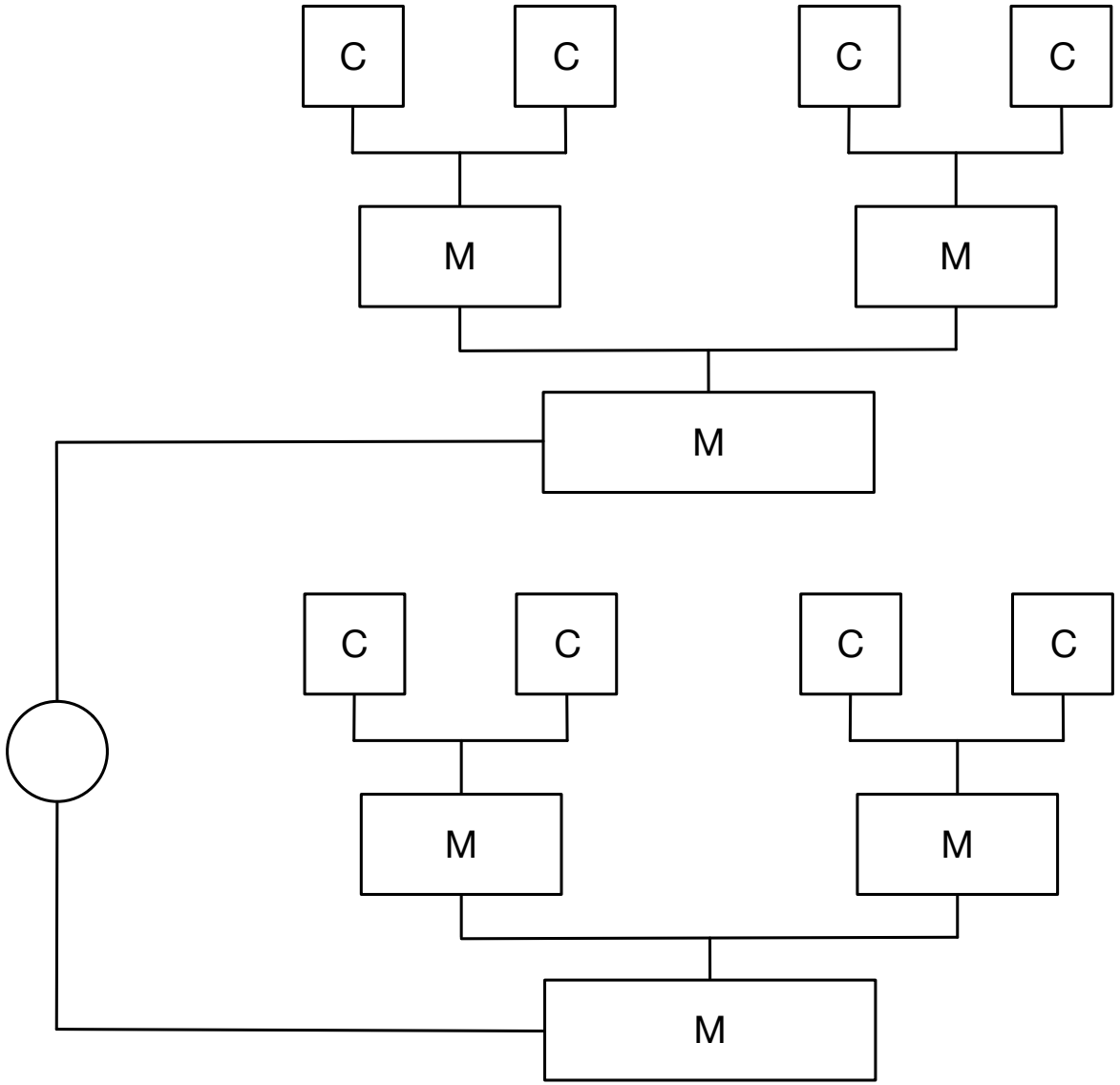
Multicore



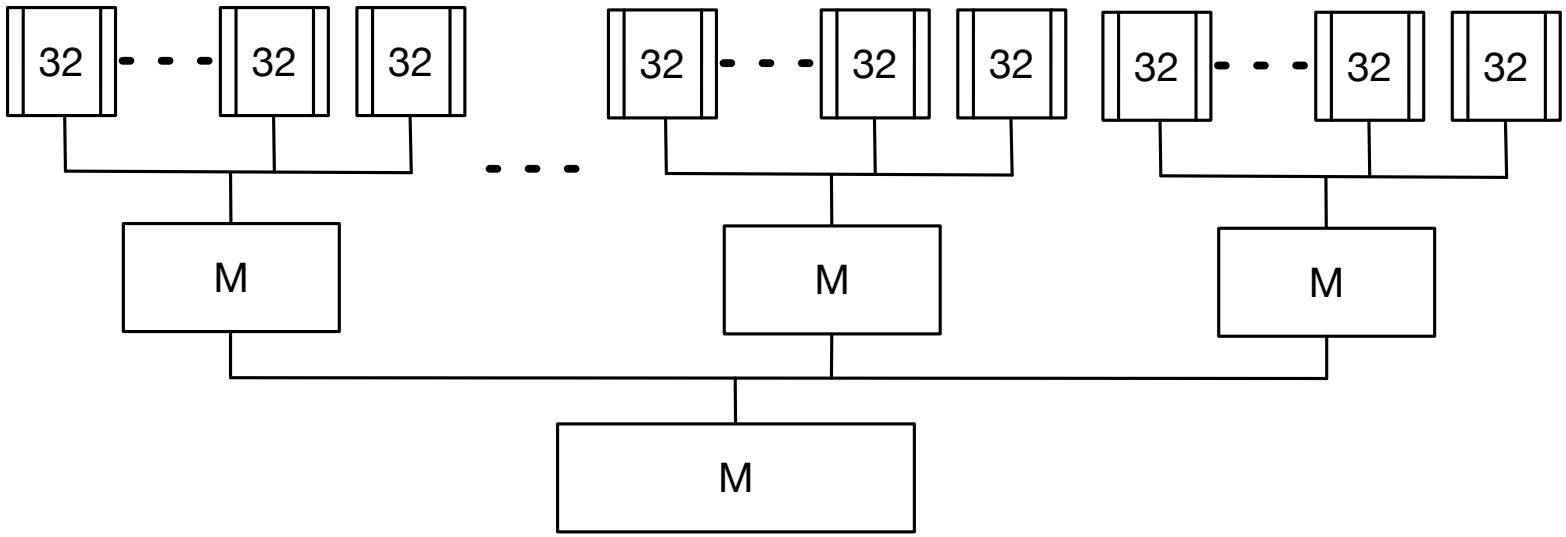
Multiprocessor or Multicomputer

- **Multiprocessor**: single address space
- **Multicomputer**: private address space per node

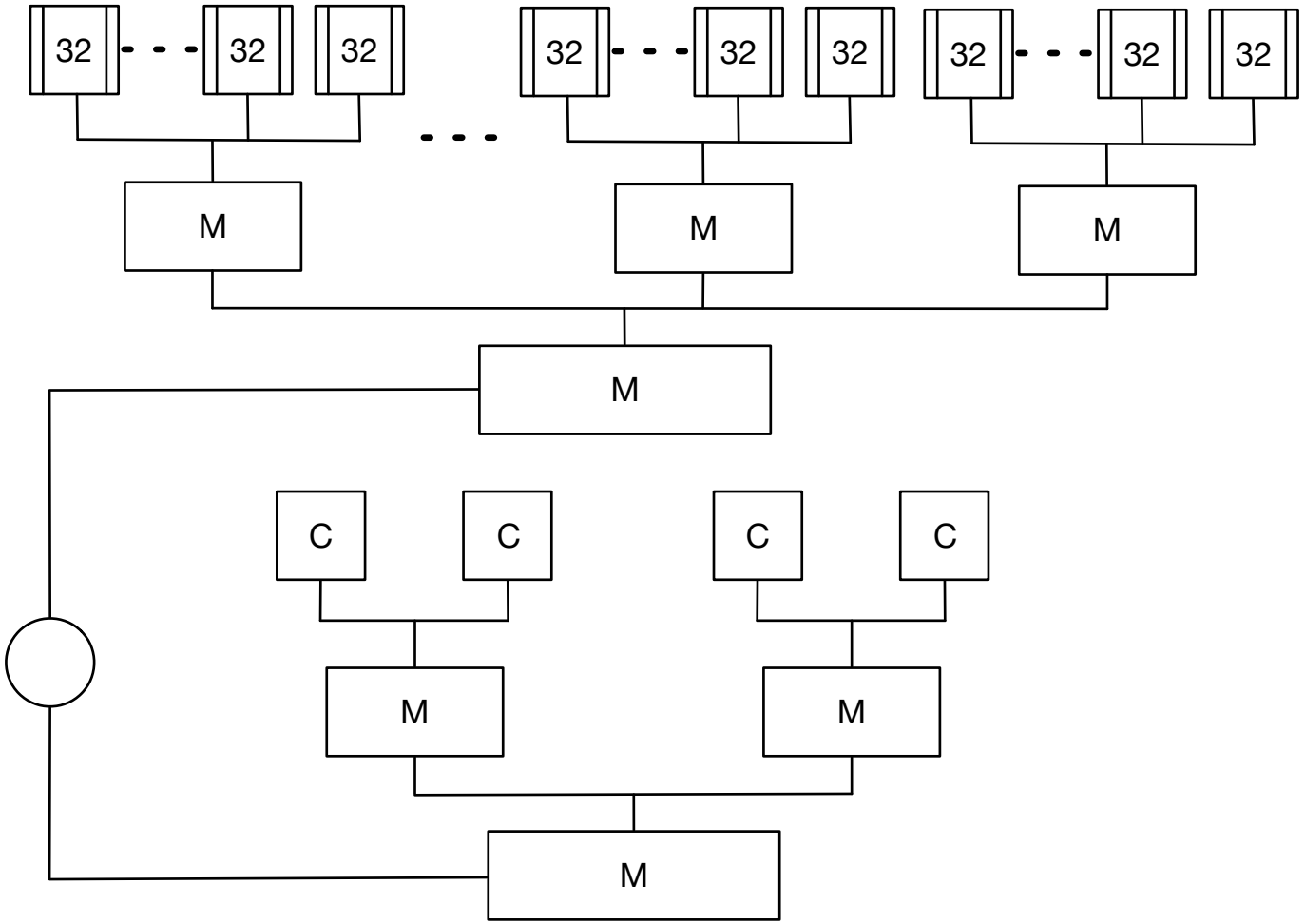
Multiprocessor or Multicomputer



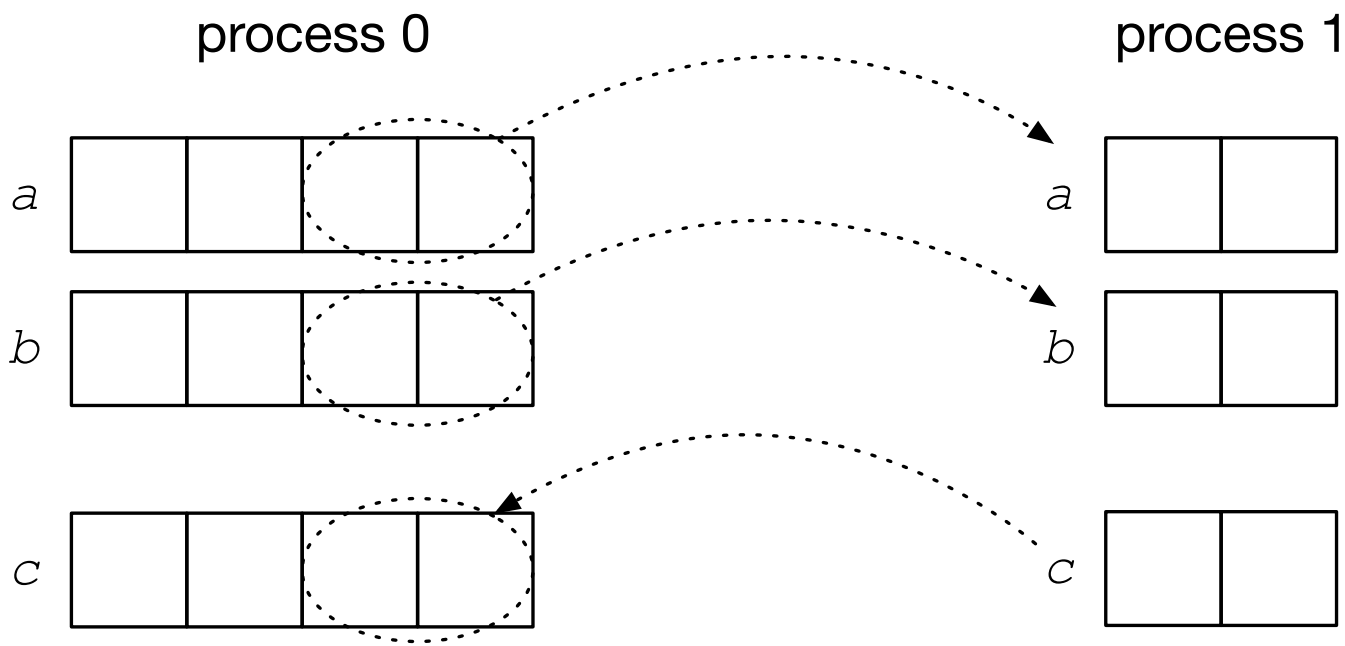
Manycore



Manycore/Multicore Hybrid



Distributed Memory Execution



Distributed Memory Execution

```
   // process 0                          // process 1
1  send a[2...3] to process 1         1  receive a[0...1] from process 0
2  send b[2...3] to process 1         2  receive b[0...1] from process 0
3  c[0] ← a[0] + b[0]                 3  c[0] ← a[0] + b[0]
4  c[1] ← a[1] + b[1]                 4  c[1] ← a[1] + b[1]
5  receive c[2...3] from process 1    5  send c[0...1] to process 0
```

Shared Memory Execution

```
// thread 0          // thread 1          // thread
                                          2
r00 ← load a[0]      r10 ← load a[1]      ...
r01 ← load b[0]      r11 ← load b[1]      ...
r01 ← add r00, r01   r11 ← add r10, r11   ...
c[0] ← store r01     c[1] ← store r11     ...
```

Not in lockstep!

Pitfalls of Shared Memory Execution

- Data races
- Memory consistency
- Cache coherence

Watch out!

```
// thread 0          // thread 1          // thread
                                          2
r00 ← load a[0]      r10 ← load a[1]      ...
r01 ← load b[0]      r11 ← load b[1]      ...
r01 ← add r00, r01   r11 ← add r10, r11   ...
c[0] ← store r01     c[1] ← store r11     ...
```

$r00 \leftarrow \text{load } c[3]$

$x \leftarrow \text{store } r00$

Data Race

One thread stores a value to a memory location while other threads load or store to the same location

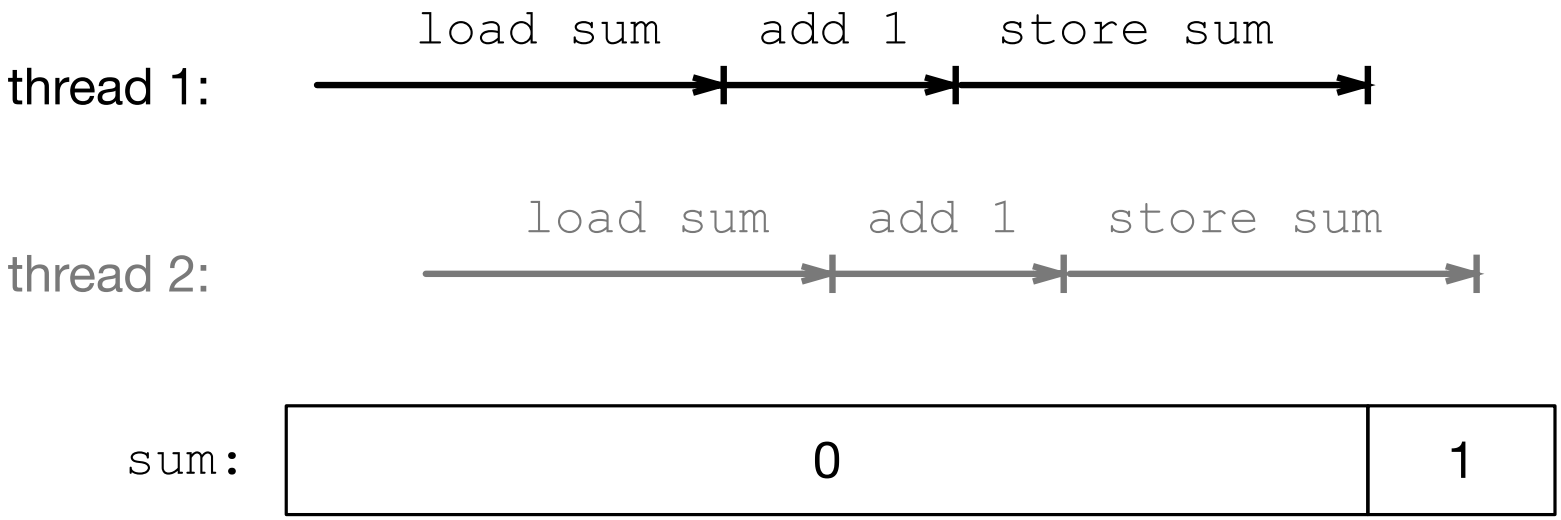
Classic Data Race Example

```
// thread 0          // thread 1
sum ← sum + 1        sum ← sum + 1
```

At machine level:

```
// thread 0          // thread 1
r00 ← load sum       r10 ← load sum
r01 ← add r00,1      r11 ← add r10,1
sum ← store r01      sum ← store r11
```

Data Race Illustration



Don't Roll Your Own Synchronization

```
// thread 0                    // thread 3
L: r00 ← load flag             c[3] ← store r31
If flag ≠ 1 goto L             flag ← store 1
r00 ← load c[3]
x ← store r00
```

Bad! Why?

Possible $c \leftarrow a + b$ Interleaving

1: $r00 \leftarrow$ load $a[0]$
2: $r10 \leftarrow$ load $a[1]$
3: $r20 \leftarrow$ load $a[2]$
4: $r30 \leftarrow$ load $a[3]$

5: $r01 \leftarrow$ load $b[0]$
6: $r11 \leftarrow$ load $b[1]$
7: $r21 \leftarrow$ load $b[2]$
8: $r31 \leftarrow$ load $b[3]$

9: $r21 \leftarrow$ add $r20, r21$
10: $r01 \leftarrow$ add $r00, r01$
11: $r31 \leftarrow$ add $r30, r31$
12: $r11 \leftarrow$ add $r10, r11$

13: $c[0] \leftarrow$ store $r01$
14: $c[1] \leftarrow$ store $r11$
15: $c[2] \leftarrow$ store $r21$
16: $c[3] \leftarrow$ store $r31$

Sequential Consistency

Each thread executes its instructions in program order. The order of the interleaving is arbitrary.

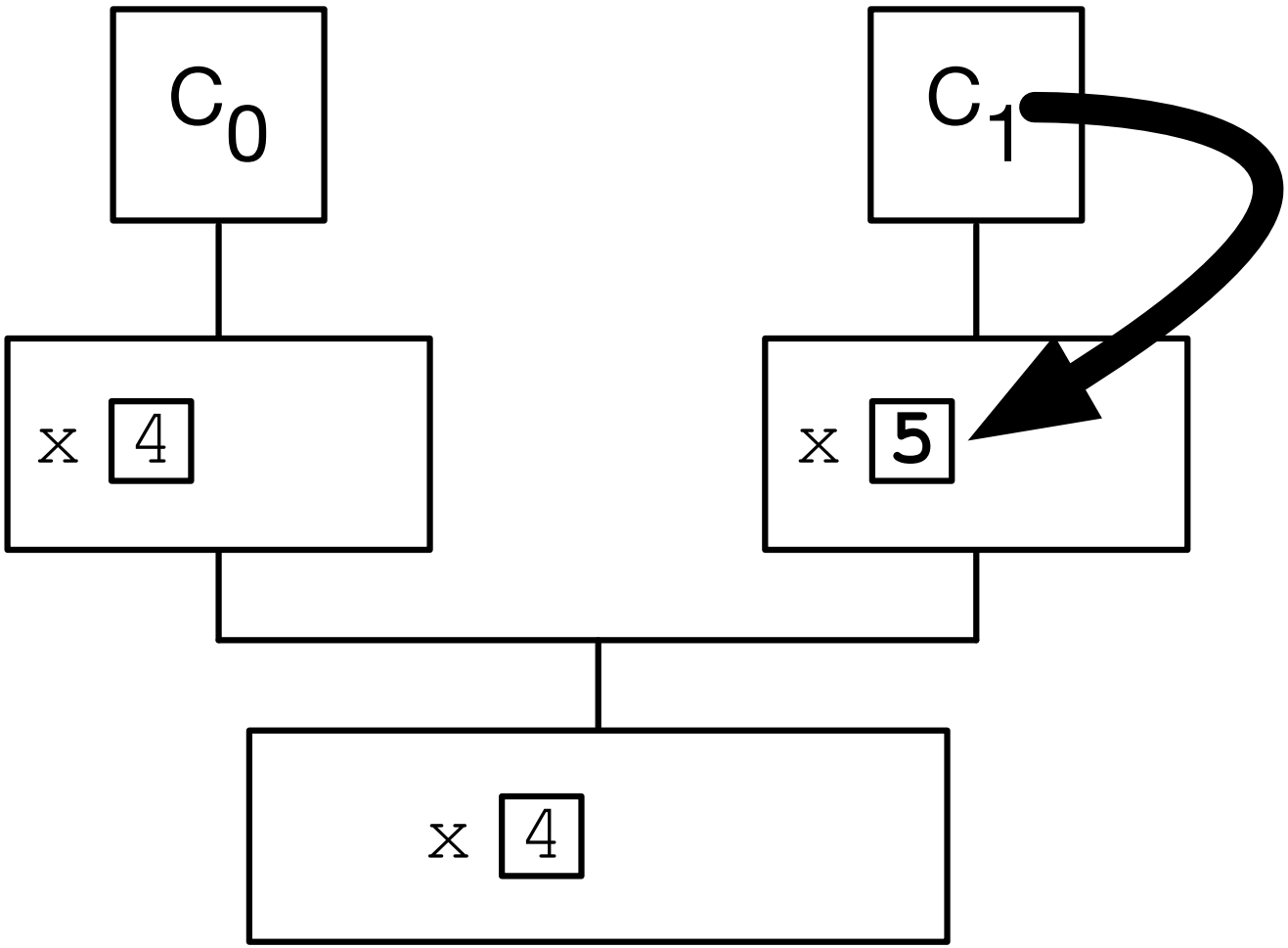
Synchronization Race

Non-sequential (relaxed) consistency:

thread 3: *flag* $\leftarrow$ store 1
thread 0: L: $r00 \leftarrow$ load
flag
thread 0: If *flag* $\neq 1$ goto L
thread 0: $r00 \leftarrow$ load $c[3]$
thread 0: $x \leftarrow$ store $r00$
thread 3: $c[3] \leftarrow$ store $r31$

If no data (or synchronization) race, sequential consistency applies

Cache Coherence Problem



Cache Coherence

At any given time either any number of cores can read or only one core can write to a *cache block*, and in addition once a write occurs it is immediately visible to a subsequent read