CSEP 524 – Parallel Computation University of Washington

Lecture 2: Parallel Architectures and Models

Michael Ringenburg
Spring 2015

Today's Topic

- Parallel hardware architectures, past and present
  - Parallel computers differ dramatically from each other
    - No standard architecture/no single programming target!
  - Parallelism introduces new costs
    - Communication
    - Resource contention
  - Ideally, details of parallel computers should be no greater concern to programmers than details of sequential computers
- How do we solve this?
  - Von Neumann model (partially) solved this for sequential computing
  - Can we come up with a similar, parallel model?

Today's Plan

- Introduce instances of basic parallel designs
  - Multicore chips
  - Symmetric Multiprocessors (SMPs)
  - Clusters
  - Multithreaded machines
- Formulate a model of computation
  - Assess the model of computation
- Bonus (?): How do we model the memory?

Today's Plan

- Introduce instances of basic parallel designs
  - Multicore chips
  - Symmetric Multiprocessors (SMPs)
  - Clusters
  - Multithreaded machines
- Formulate a model of computation
  - Assess the model of computation
- Bonus (?): How do we model the memory?

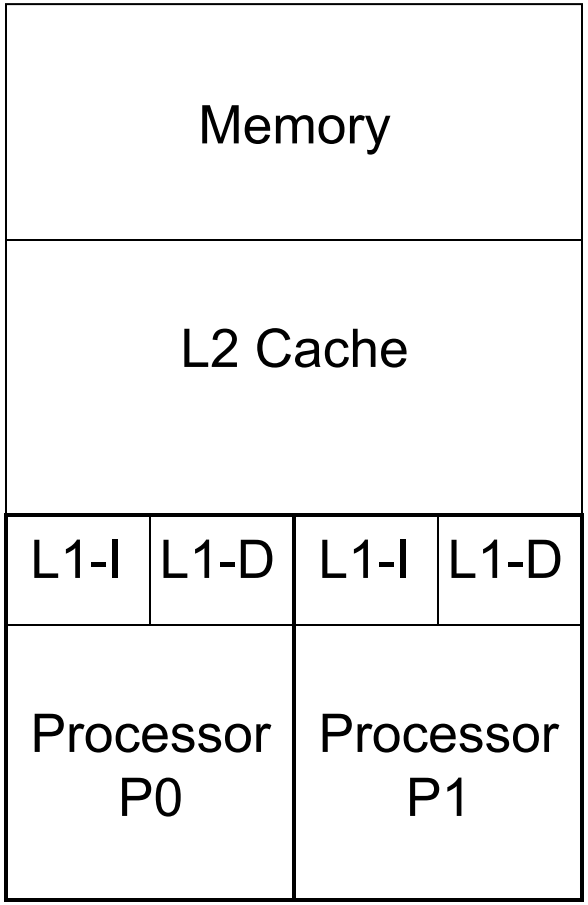
One of the hardest parts of parallel computing …

Multi-core Chips

- Multi-core means more than one processor per chip
  - Consequence of Dennard Scaling failing to keep pace with Moore's Law scaling
- Main advantage: More ops per tick
- Main disadvantages: Programming, BW
- Early examples: IBM's PowerPC 2002, AMD Dual Core Opteron 2005, Intel CoreDuo 2006
  - We'll discuss AMD and Intel variations

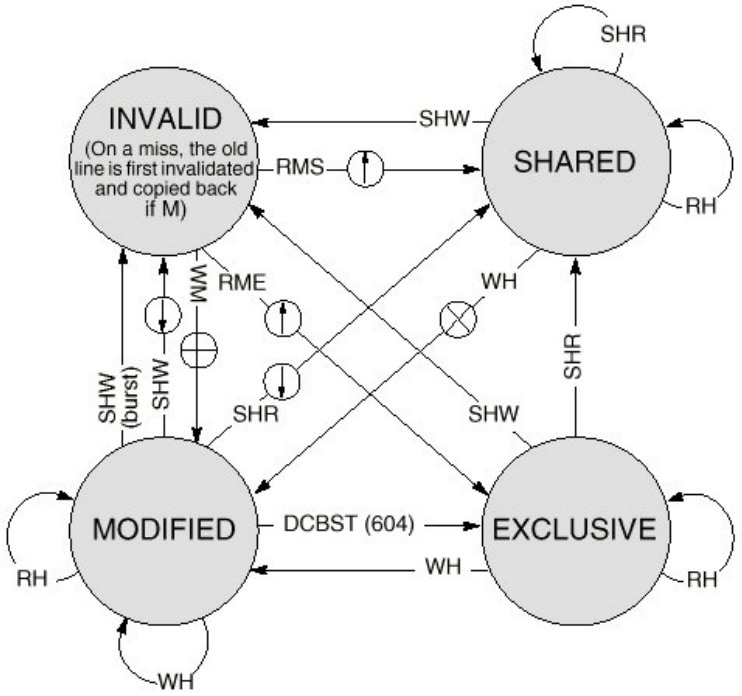
Intel CoreDuo (2006)

- 2 32-bit Cores
- Private 32K L1 cache per core
  - Separate instruction and data
- Shared 2 or 4 MB L2
- MESI cache coherence protocol
  - See next slide



MESI Protocol

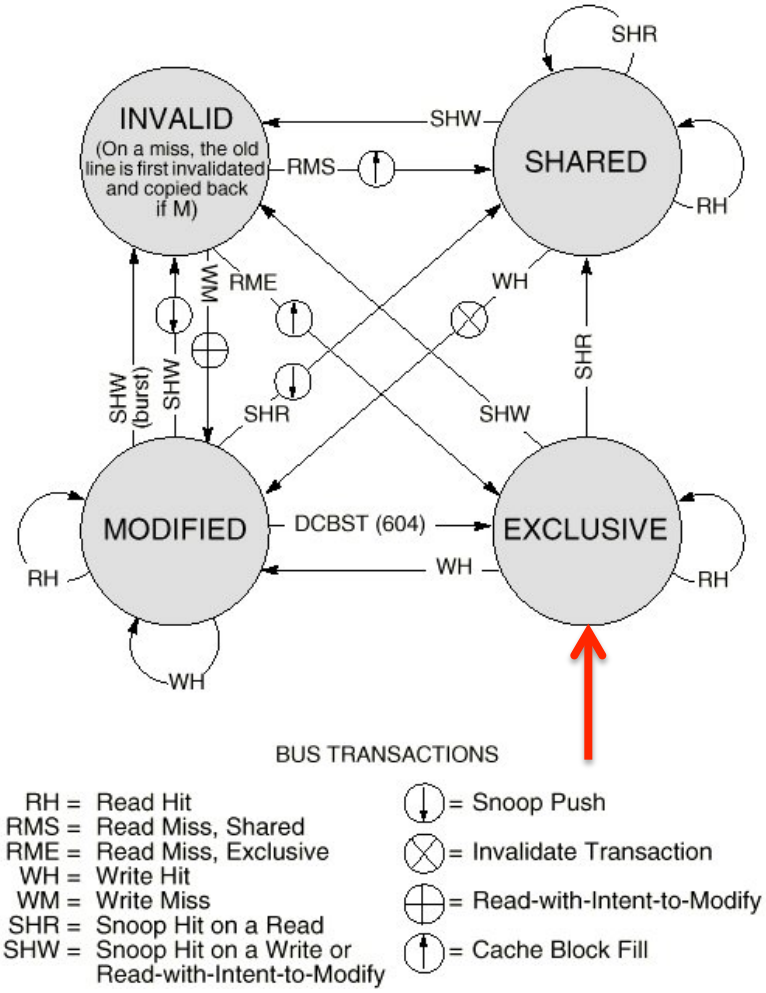
- Standard Protocol for cache - coherent shared memory
  - Mechanism for multiple caches to give single memory image
  - Complex, but cool …



Thanks: Slater & Tibrewala of CMU

MESI Protocol

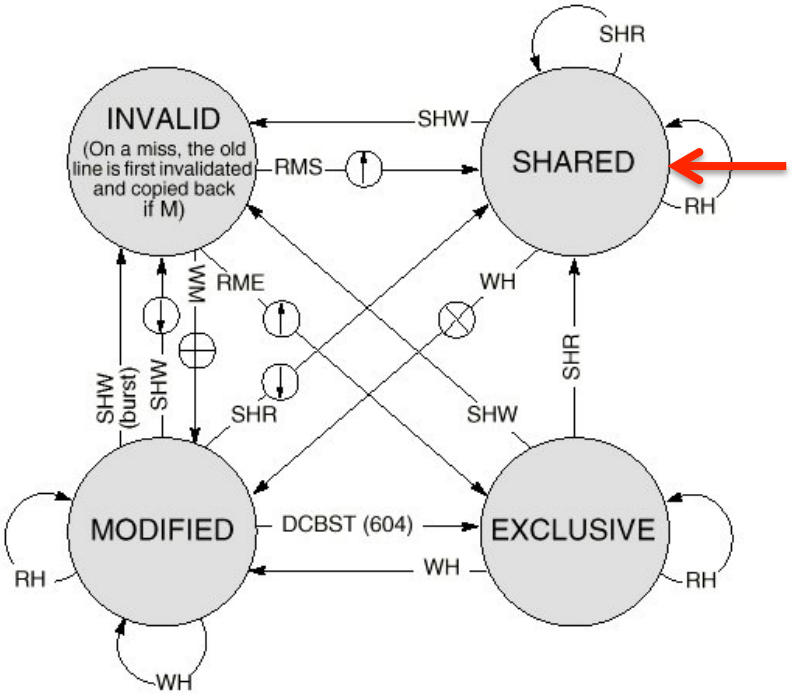
- Modified-Exclusive-Shared-Invalid
- Upon loading, a line is marked Exclusive (E)
- Subsequent reads by same core are OK
  - State unchanged

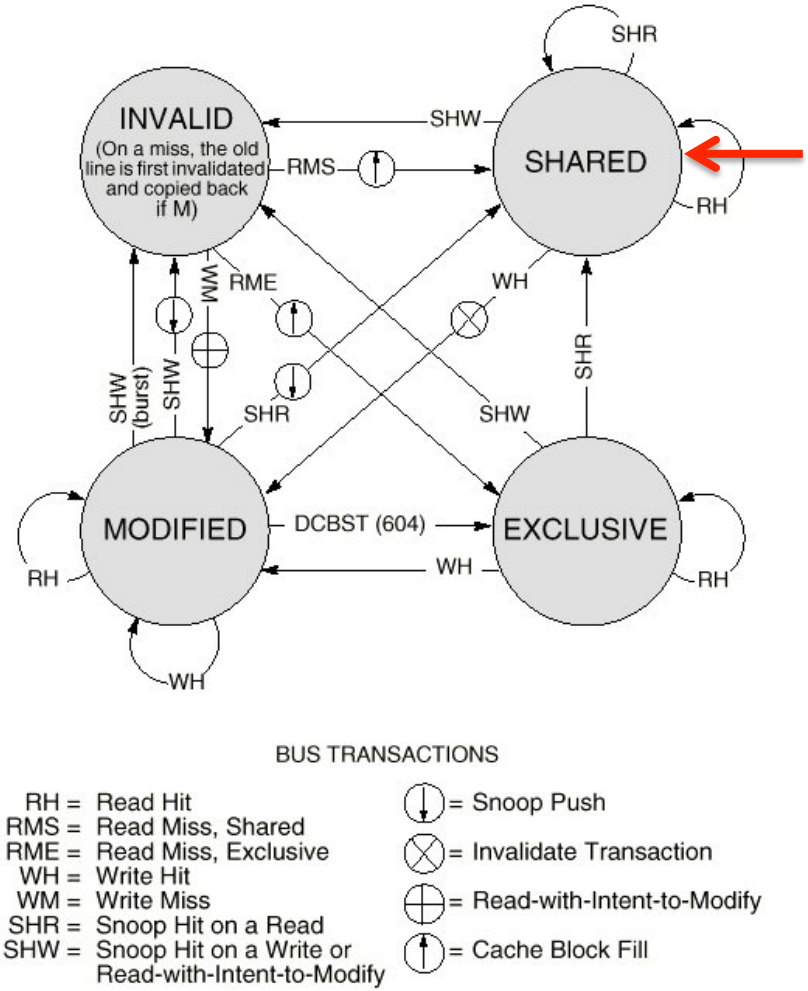


Thanks: Slater & Tibrewala of CMU

MESI Protocol

- If another core reads the same line, mark it as Shared (S)
  - In both caches

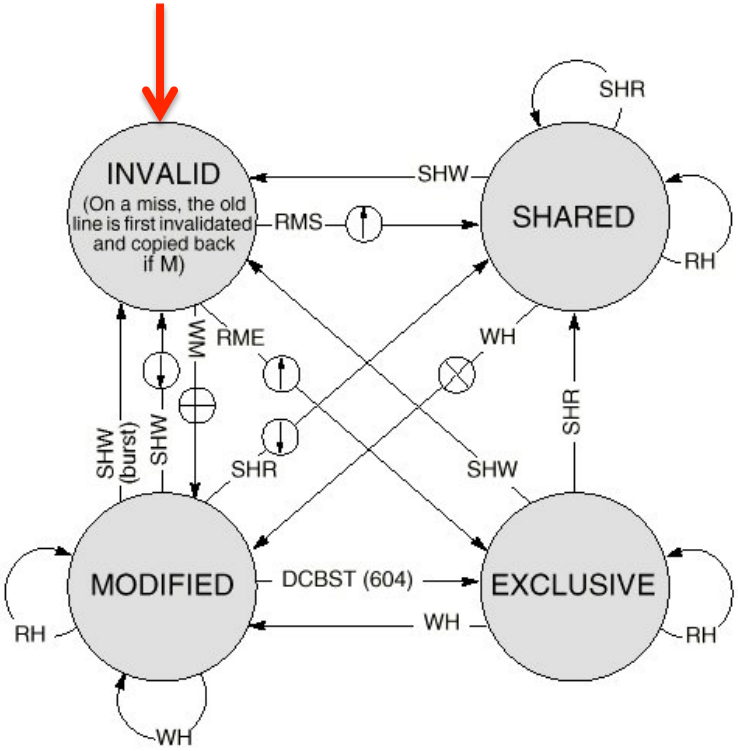


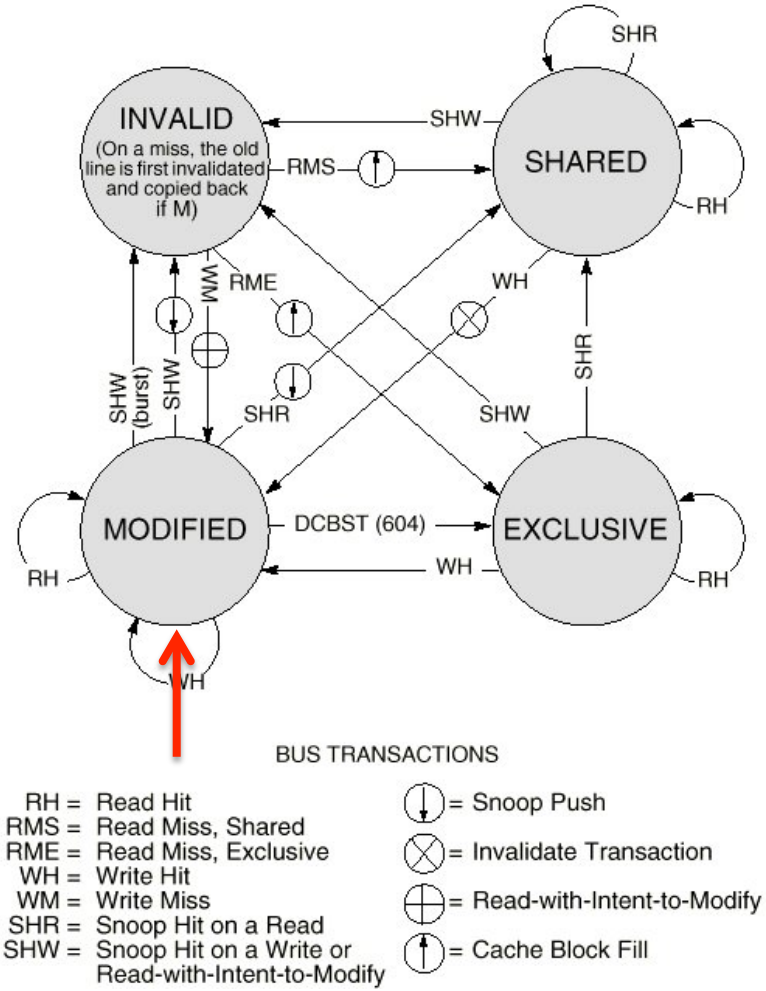


Thanks: Slater & Tibrewala of CMU

MESI Protocol

- If a core write the line, mark it as Modified (M).
  - If it is shared, mark it as Invalid (I) in other caches.

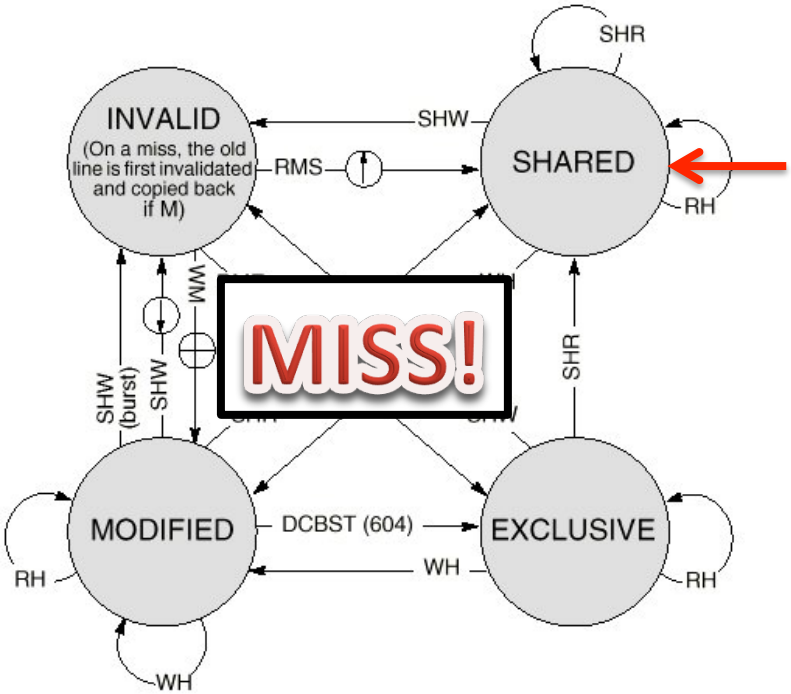


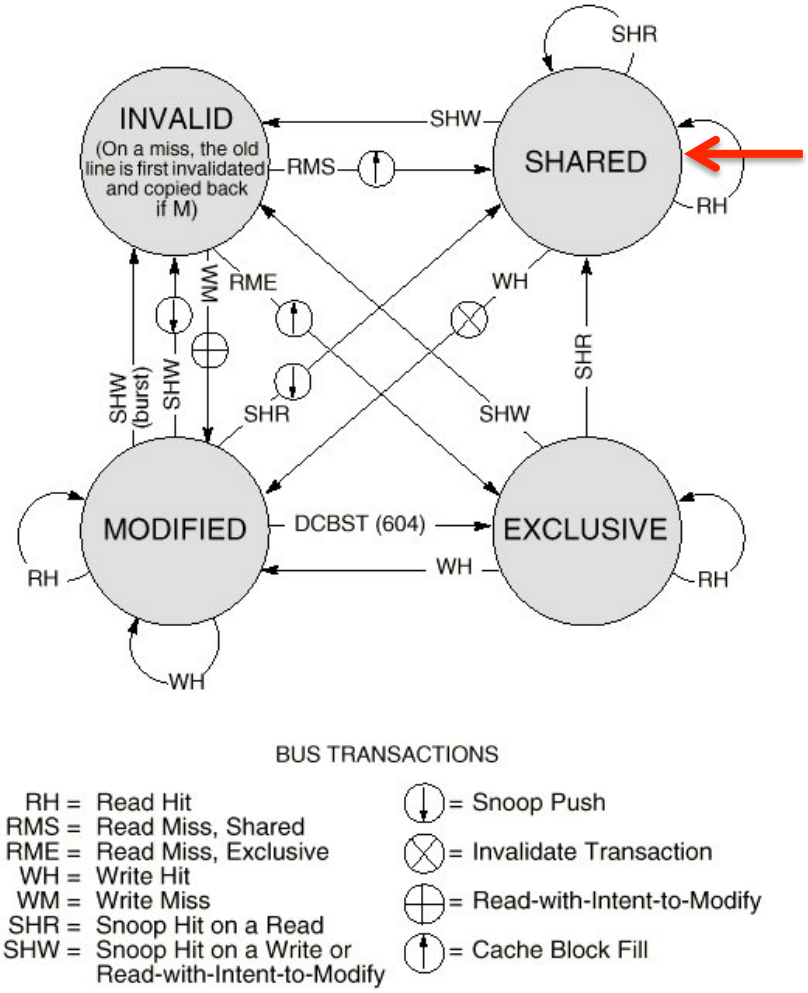


Thanks: Slater & Tibrewala of CMU

MESI Protocol

- Access to an Invalid (I) results in a cache miss.
- Also detected by Modified (M) core, causing it to write back and switch to Shared (S)





Thanks: Slater & Tibrewala of CMU

Example

Thread 1

```
for (int i = 0; i < n; i++) {
  condition = foo();
  if (condition) {
    counter[0]++;
  }
}
```

Thread 2

```
for (int i = 0; i < n; i++) {
  condition = foo();
  if (condition) {
    counter[1]++;
  }
}
```

(Assume counter[0] and counter[1] are on the same cache line)

	Core 1 state	Core 2 state
Initial	I	I
Core 1 Load		
Core 2 Load		
Core 1 Store		
Core 2 Store		
(cont.)		
(cont.)		
Core 2 Eviction		
Core 2 Load		
Core 2 Store		
Core 2 Load		

Example

Thread 1

```
for (int i = 0; i < n; i++) {
  condition = foo();
  if (condition) {
    counter[0]++;
  }
}
```

Thread 2

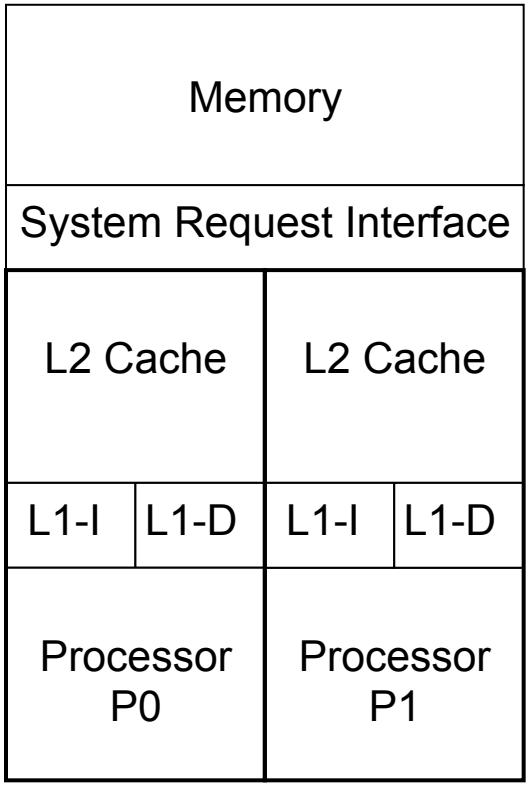
```
for (int i = 0; i < n; i++) {
  condition = foo();
  if (condition) {
    counter[1]++;
  }
}
```

(Assume counter[0] and counter[1] are on the same cache line)

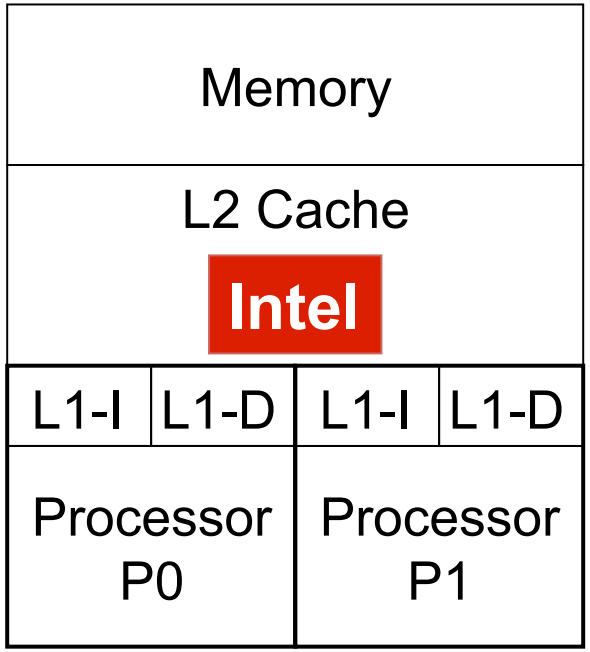
	Core 1 state	Core 2 state
Initial	I	I
Core 1 Load	Load, E	I
Core 2 Load	S	Load, S
Core 1 Store	M	I
Core 2 Store	Writeback, S	I
(cont.)	S	Load, S
(cont.)	I	M
Core 2 Eviction	I	Writeback, I
Core 2 Load	I	E
Core 2 Store	I	M
Core 2 Load	I	M

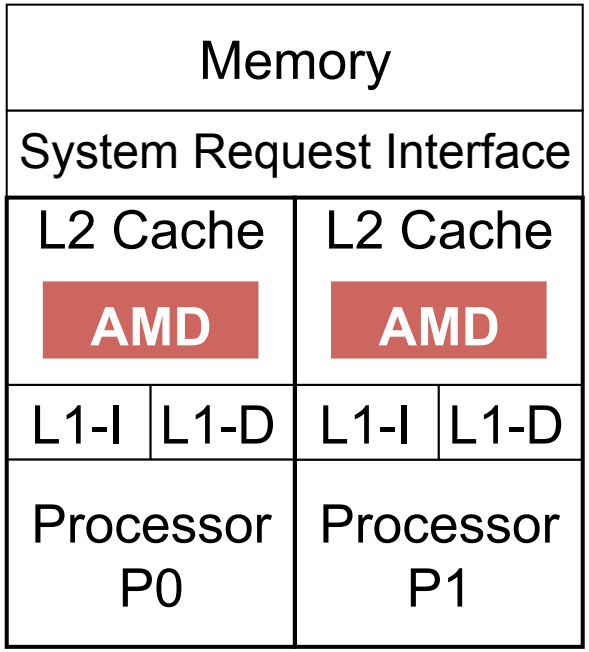
AMD Dual Core Opteron ‘05

- 2 64-bit Opterons
- 64K private L1s
- 1 MB private L2s
  - Key difference between these early architectures
- SRI handles cache coherence, main memory transfers
- MOESI cc-protocol
  - O = Owned. Like shared, but with exclusive modification rights, and responsible for supplying value to other caches
  - Allows multiple caches to share a “dirty” value



Comparing Core Duo/ Dual Core

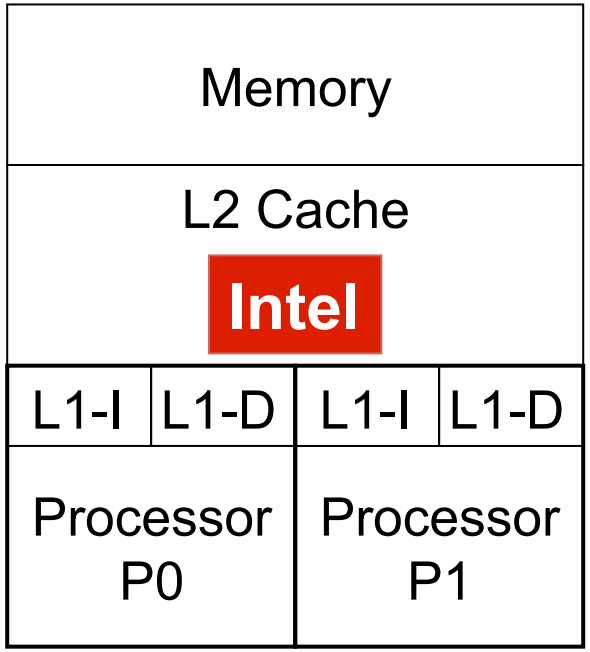


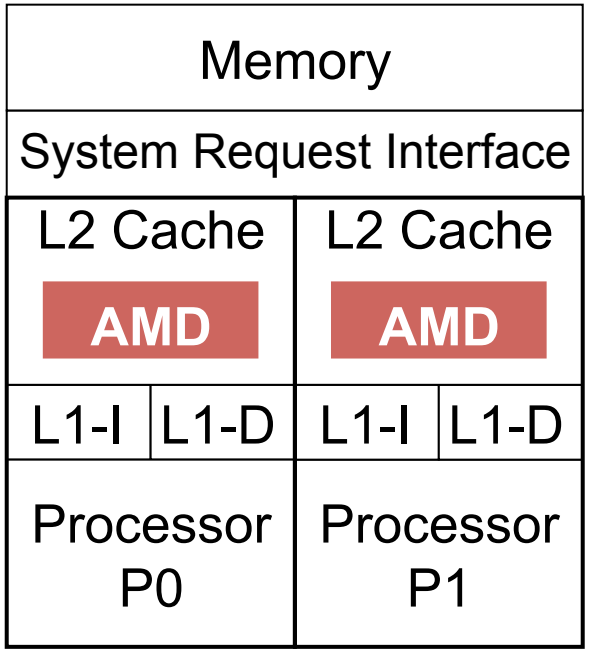


- Advantages and disadvantages of each?
- Which would work better if we had multiple chips connected to a shared memory?

(These diagrams elide some important pieces, like buses)

Comparing Core Duo/ Dual Core

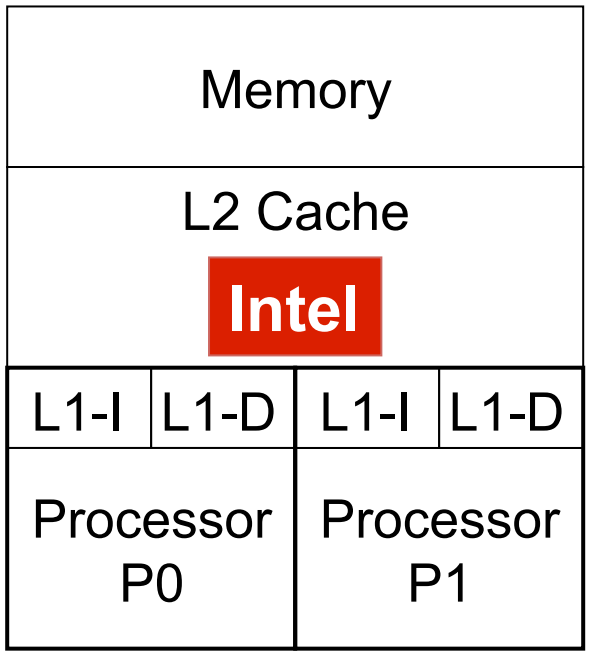


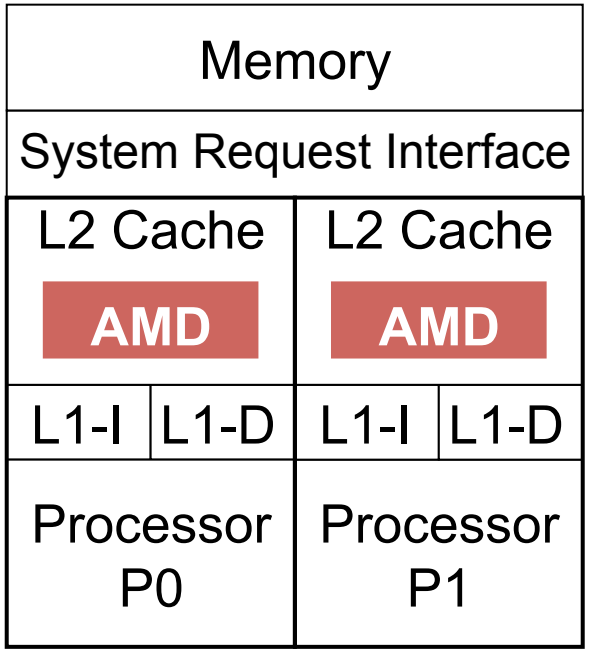


- AMD Dual Core: Larger private memory, coherence managed at the "back" enables easier sharing with others, 'O' state reduces write backs

(These diagrams elide some important pieces, like buses)

Comparing Core Duo/ Dual Core

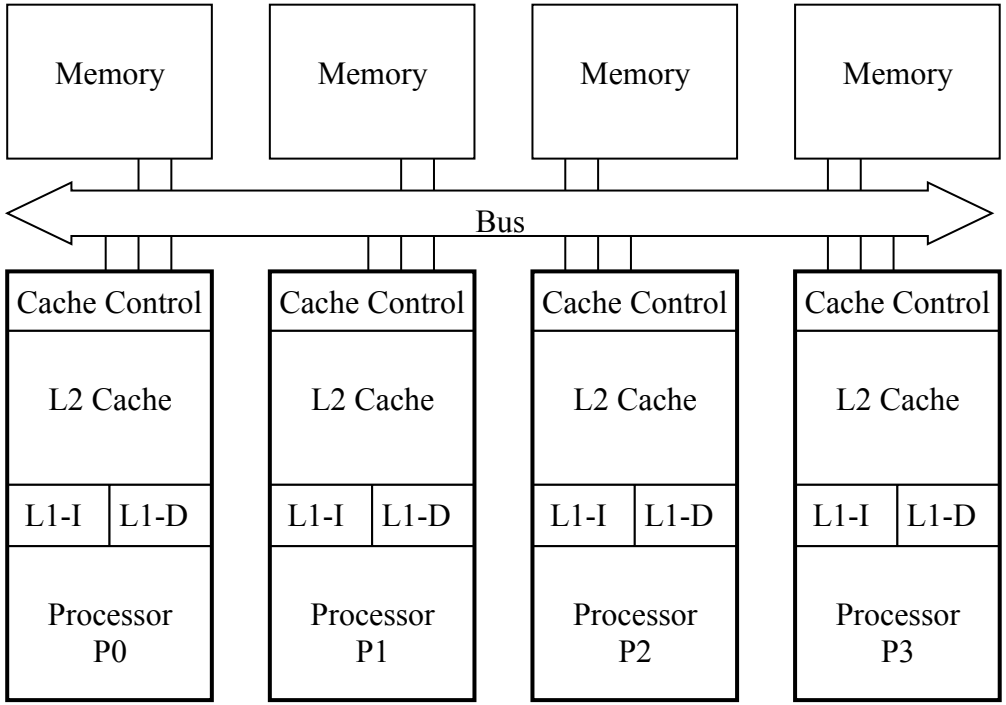




- Intel Core Duo: coherence managed closer to cores allows faster communication between cores, full L2 available to single-threaded code

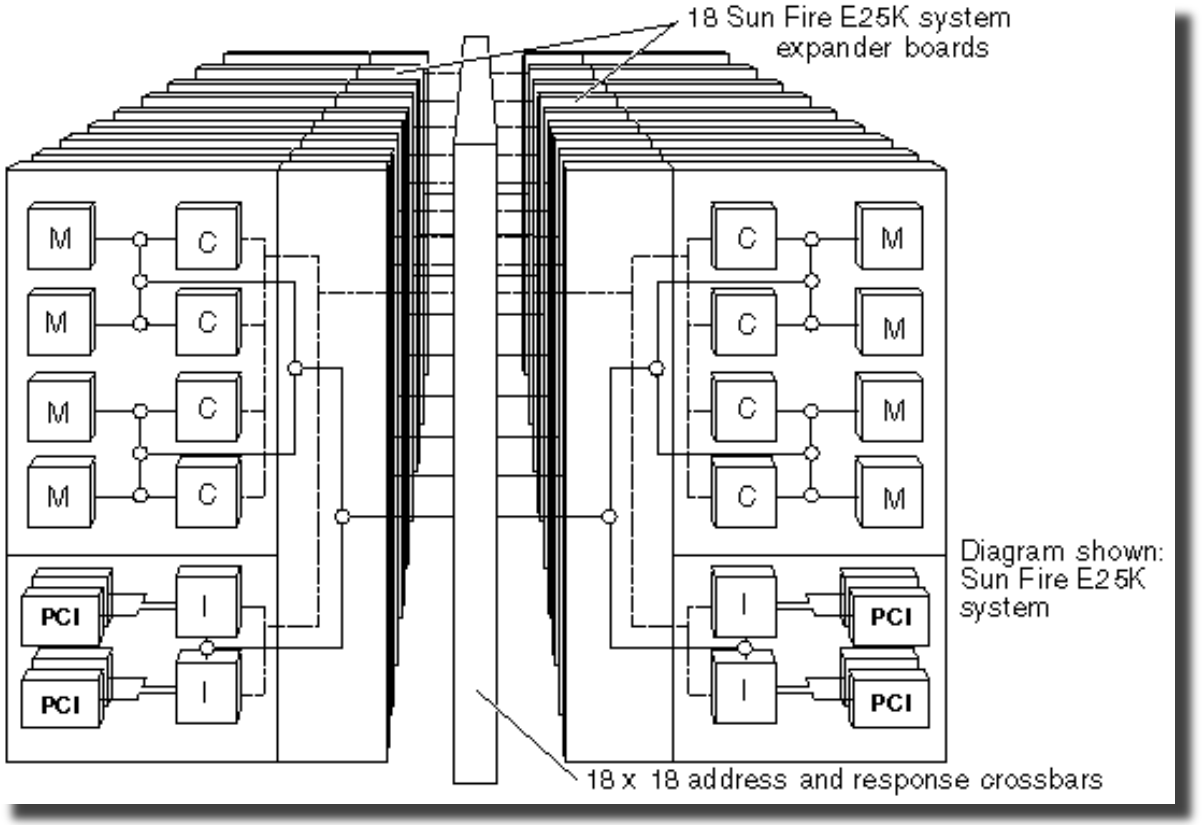
(These diagrams elide some important pieces, like buses)

Symmetric Multiprocessors (SMPs)



- All processors attached to a single large shared memory
  - (Individual DIMMs may be physically located near different processors)
- Consistent memory view via common connection to memory that “snoops” on other processors’ accesses

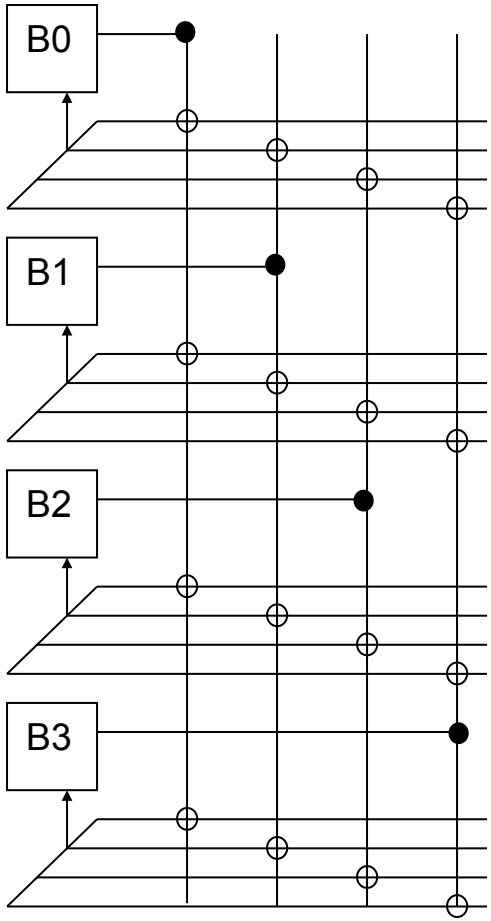
Sun Fire E25K



- Up to 18 four-processor boards, connected by crossbar switch (all boards directly connected to each other)
  - Snoopy bus on each board
  - Global directory (tracks views of memory) for inter-board coherence

Cross-Bar Switch

- Maintaining coherence requires low latency connections between every pair of processors
- A crossbar is a network directly connecting each processor (or board) to every other processor (or board)
- Crossbars grow as n^2 (where n = # of connections), making them impractical for large n



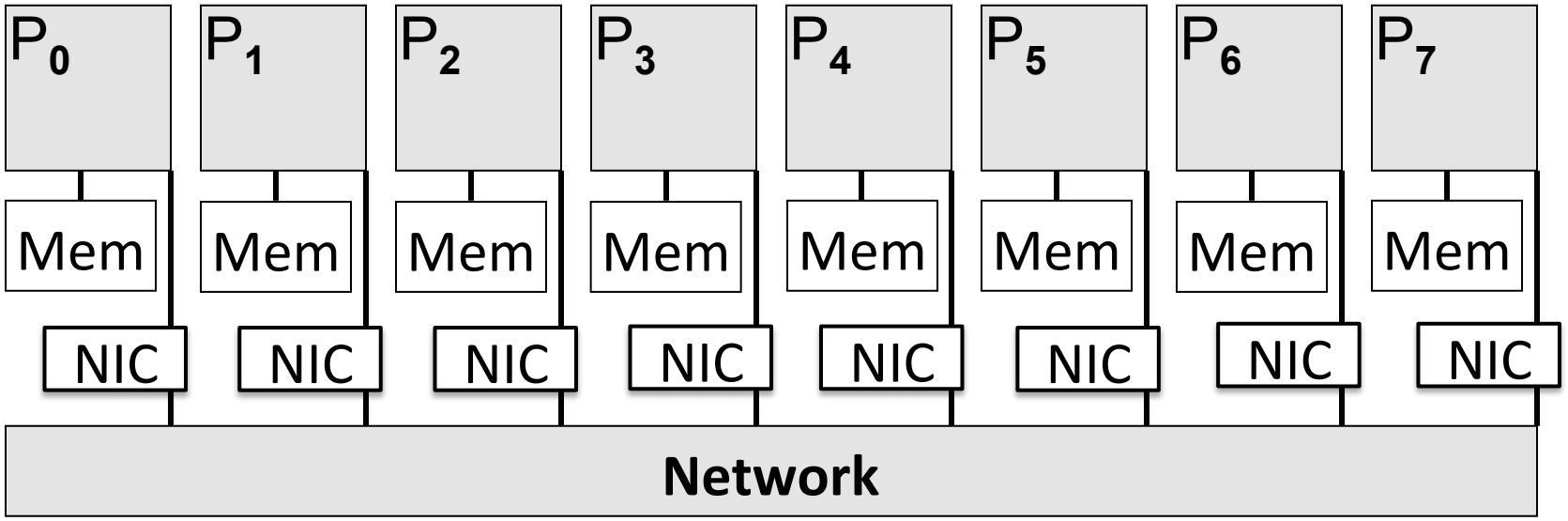
Sun Fire E25K: The Limit?

- X-bar gives low latency for snoops allowing for shared memory
- 18 x 18 X-bar is basically the limit
- Raising the number of processors per board will, on average, increase congestion
- Huge amount of complex engineering required to make 72 processor SMP feasible
- So, how could we make a larger machine?

Sun Fire E25K: The Limit?

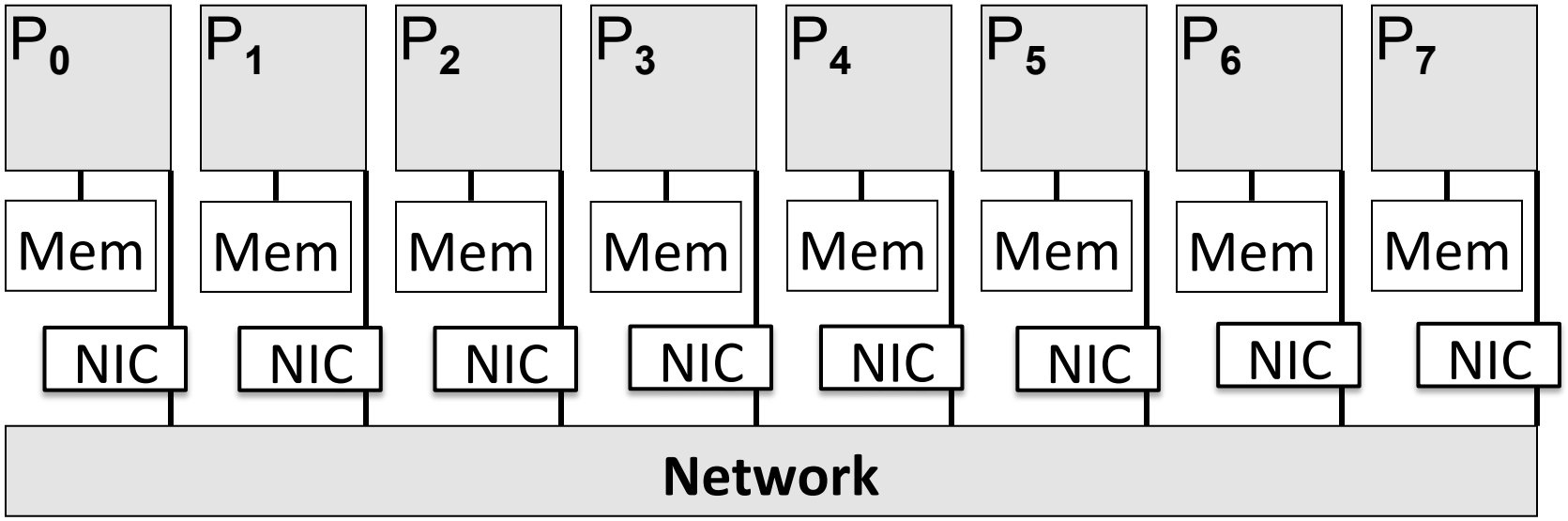
- X-bar gives low latency for snoops allowing for shared memory
- 18 x 18 X-bar is basically the limit
- Raising the number of processors per board will, on average, increase congestion
- Huge amount of complex engineering required to make 72 processor SMP feasible
- So, how could we make a larger machine?
  - **Dispense with shared memory…**

Clusters



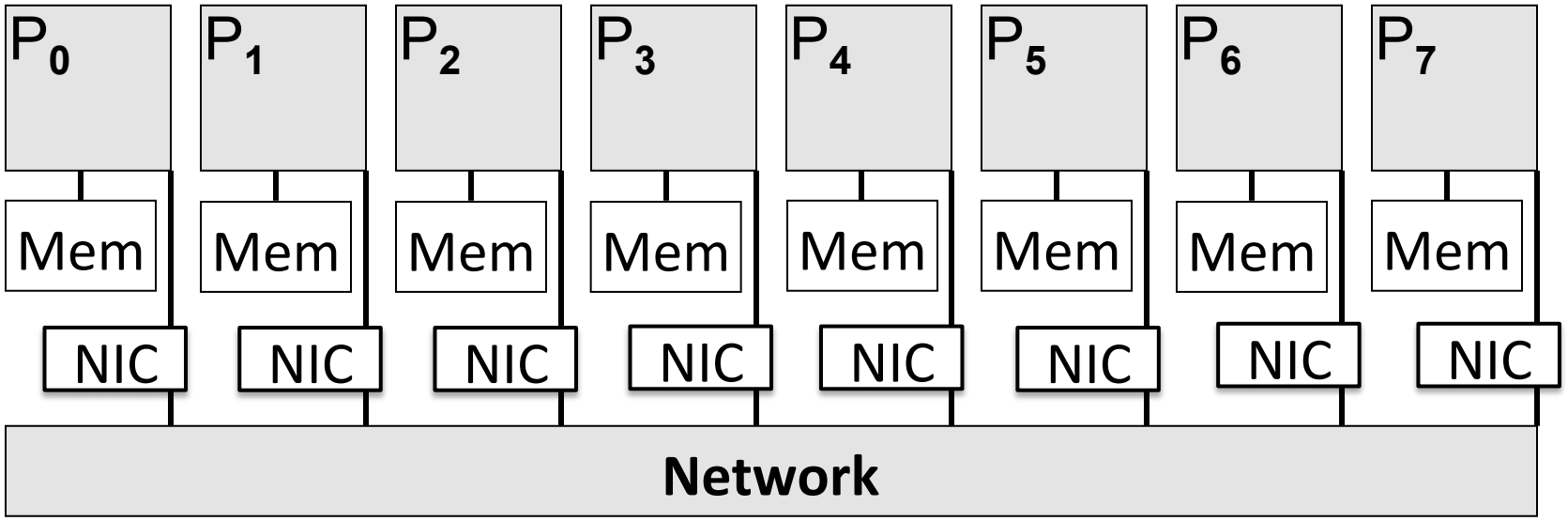
- Commodity servers, connected via a network ("interconnect").
  - Each server typically has it's own disk(s) and memory
  - Servers are often blades inside of a single rack, rather than separate boxes
- Often programmed using message passing (e.g., MPI), or frameworks like Hadoop or Spark

Clusters



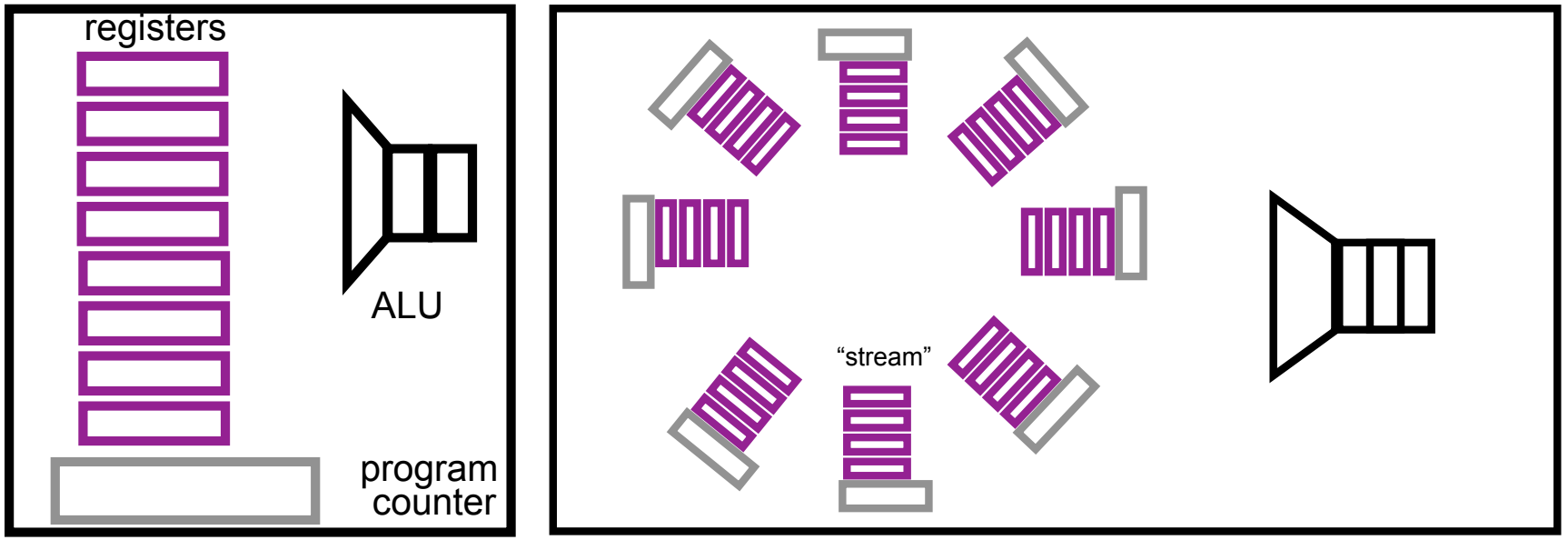
- Common cluster interconnects:
  - Ethernet: Cheap, but higher latencies, less bandwidth (new standards coming, though).
  - Infiniband: Lower latency, RDMA support, open standard
  - Custom/proprietary (e.g., Cray Aries: high global bandwidth, low diameter topology – scales to huge systems)

Clusters/Supercomputers



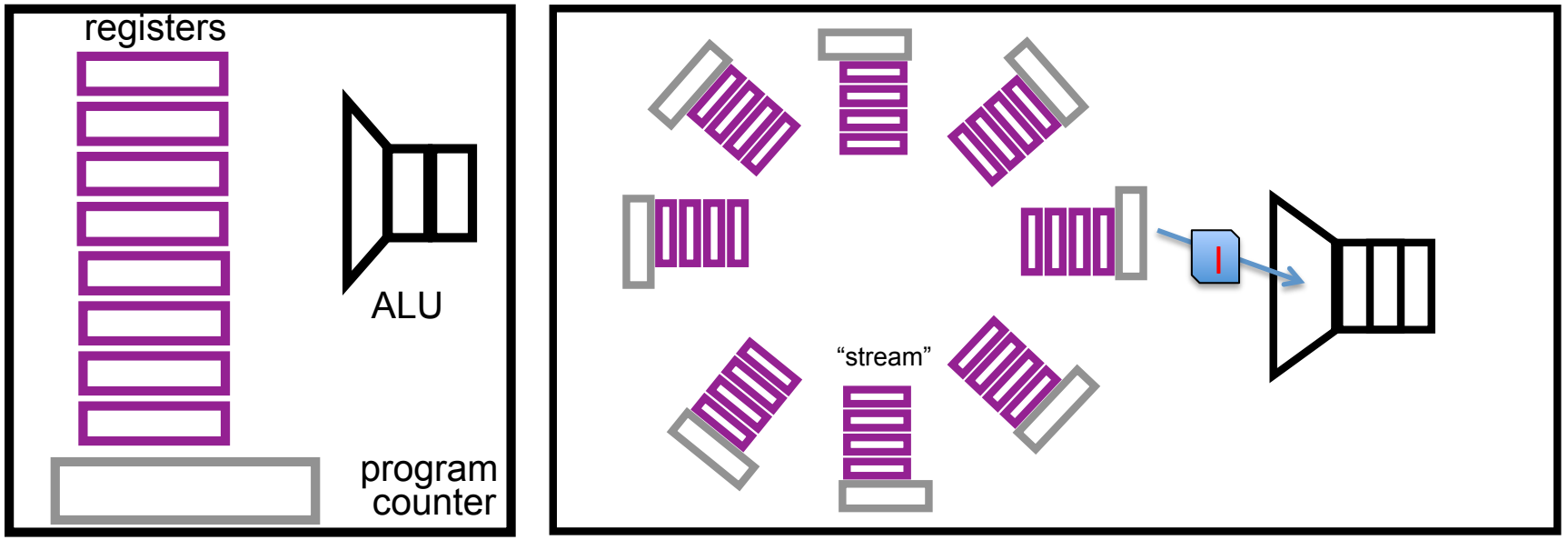
- Boundary between clusters and supercomputers is blurring… many supercomputers are essentially clusters that have/are:
  - Well provisioned nodes/blades (e.g., large memory, multiple sockets, high core count)
  - Tightly connected (high bandwidth, low latency, low diameter network)
  - Built to scale to very large node counts (see previous)
  - Often custom software (e.g., stripped down OS, custom compiler/PE, system management)

Multithreaded machines: Cray MTA/XMT



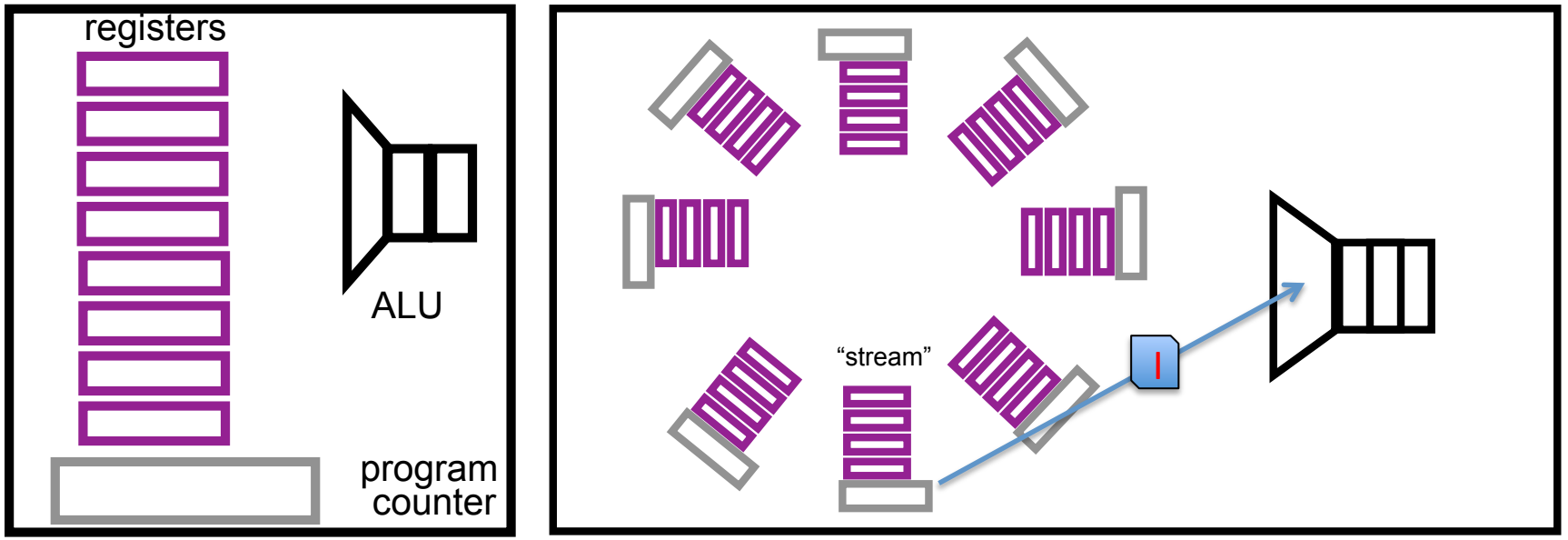
- Threadstorm processor – 128 threads/processor, each with own register bank (including PC)
  - Every clock cycle execute an instruction from a different (unblocked) thread
  - At most one instruction in pipeline (21 stages) from any thread at any time
- Thus, a thread will execute an instruction every 21-128 cycles. Why do this? Hides memory latency (provided there is sufficient parallelism).

Cray MTA/XMT



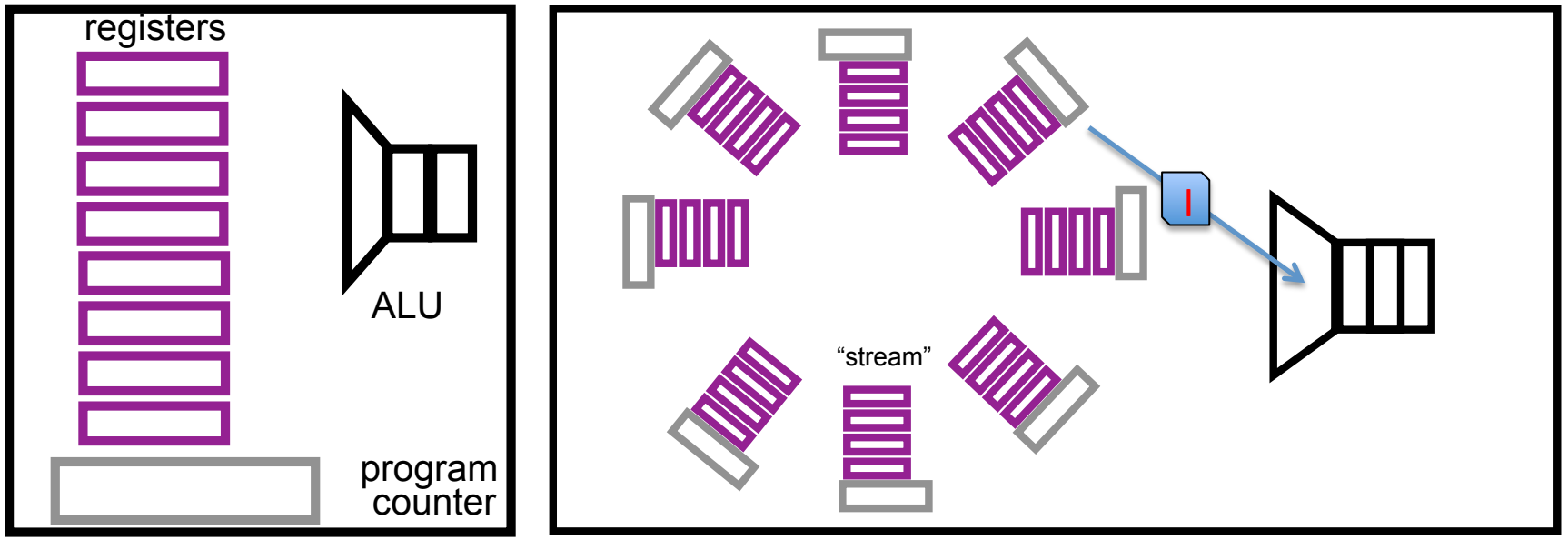
- Threadstorm processor – 128 threads/processor, each with own register bank (including PC)
  - Every clock cycle execute an instruction from a different (unblocked) thread
  - At most one instruction in pipeline (21 stages) from any thread at any time
- Thus, a thread will execute an instruction every 21-128 cycles. Why do this? Hides memory latency (provided there is sufficient parallelism).

Cray MTA/XMT



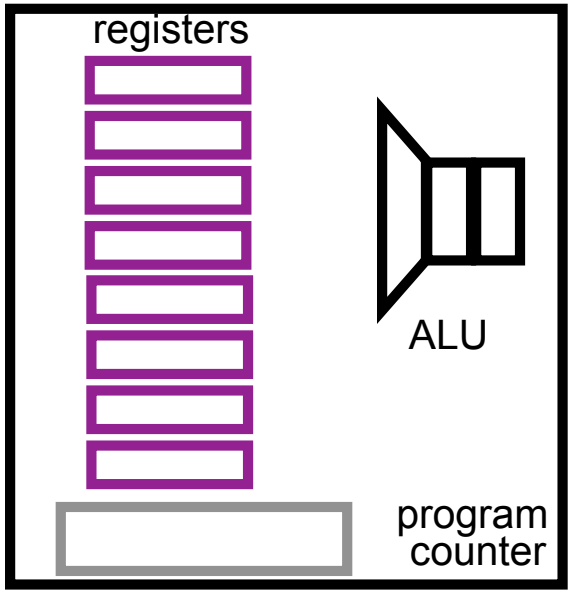
- Threadstorm processor – 128 threads/processor, each with own register bank (including PC)
  - Every clock cycle execute an instruction from a different (unblocked) thread
  - At most one instruction in pipeline (21 stages) from any thread at any time
- Thus, a thread will execute an instruction every 21-128 cycles. Why do this? Hides memory latency (provided there is sufficient parallelism).

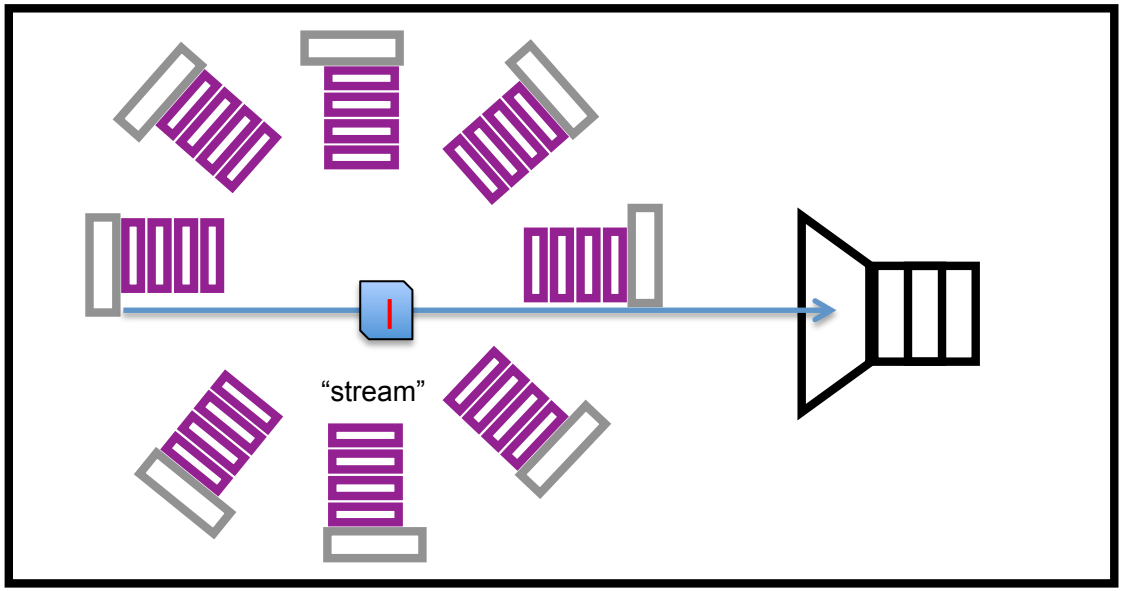
Cray MTA/XMT



- Threadstorm processor – 128 threads/processor, each with own register bank (including PC)
  - Every clock cycle execute an instruction from a different (unblocked) thread
  - At most one instruction in pipeline (21 stages) from any thread at any time
- Thus, a thread will execute an instruction every 21-128 cycles. Why do this? Hides memory latency (provided there is sufficient parallelism).

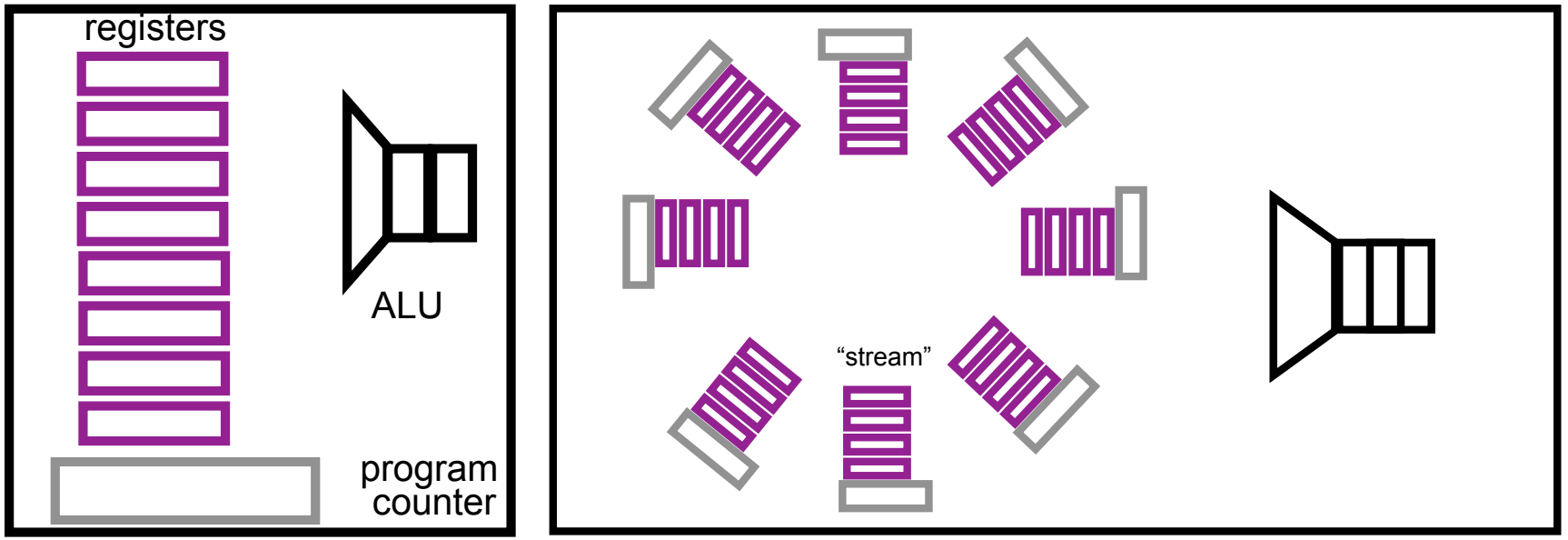
Cray MTA/XMT





- Threadstorm processor – 128 threads/processor, each with own register bank (including PC)
  - Every clock cycle execute an instruction from a different (unblocked) thread
  - At most one instruction in pipeline (21 stages) from any thread at any time
- Thus, a thread will execute an instruction every 21-128 cycles. Why do this? Hides memory latency (provided there is sufficient parallelism).

Cray MTA/XMT



- Requires high degree of parallelism to keep pipelines full
  - Fast context switches and built-in synchronization primitives allows very fine-grained parallelism (individual loop iterations)
  - Parallelizing compiler
  - Hashed global memory makes memory "appear" uniform – simplifies efficient parallelization

Co-Processor Architectures

- A powerful parallel design is to add coprocessors/ accelerators to standard design
  - Graphics Processing Units – massively parallel floating point computations
  - Cell Processor - multiple vector units
  - Attached FPGA chip(s) - compile to a circuit
- Have all proven difficult to program – manage when to move data back and forth.
- New trend being discussed “on-chip” FPGAs and other accelerators
  - Related to Dark Silicon trend – a way to use those extra transistors/board area
  - Depending on implementation, may eliminate or reduce cost of data transfers.

The Parallel Programming Problem

- Huge variety of architectures
  - We just sampled a few – barely scraped the surface…
- How can we understand what is important, and abstract away the rest?
- Is there any hope for "universal parallel programs"?
- Or (perhaps more realistically), programs that work "reasonably well" on "most parallel architectures"?
  - Can potentially be further tuned/refined for specific machines/architectures
  - Similar to sequential programming?

Some Options for Solving the PPP

- Leave the problem to the compiler …
  - Discussed this last week – compilers can help with localized parallelization, not fundamental algorithm rewrites
- Adopt an abstract parallel language that can target to any platform
  - Will programmers be willing to learn new language?
  - What is the right level of abstraction?
  - Cray's Chapel (guest lecture in 3 weeks) is a good example
    - Also X10 from IBM, Fortress from Sun (cancelled in 2012)

More Options for Solving the PPP

- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code
  - To work with multiple languages, limit base language assumptions
  - Libraries use a specific interface (function call) limiting possible syntactic abstractions (e.g., new types of loops)
  - Achieving consistent semantics is difficult
  - Examples: MPI, Pthreads
- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up …
  - Not a full solution until languages are available
  - The solution works in sequential world (von Neumann model)
  - Requires discovering (and predicting) what the common capabilities are
  - Solution needs to be (continually) validated against actual experience
  - We'll be discussing one such model next …

Summary of Options for PPP

- Leave the problem to the compiler …
- Adopt an abstract language that can target to any platform …
- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code …
- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up …
- What are your thoughts??

Why is Sequential Programming Successful?

When we write programs in C they are ...

- Efficient -- programs run fast, especially if we use performance as a goal
  - E.g., traverse arrays in row major order to improve caching
- Economical -- use resources well
  - E.g., represent data by packing memory
- Portable – Efficient programs usually run reasonably well on any computer with C compiler
- Easy to write -- we know many 'good' techniques
  - reference data, don't copy

These qualities all derive from von Neumann model

Von Neumann (RAM) Model

- Call the ‘standard’ model of a random access machine (RAM) the von Neumann model
  - A processor executing one basic operation at a time (3-address code)
  - PC pointing to the next instruction of program in memory
  - “Flat”, randomly accessed memory requires 1 **time unit** (*not* clock cycle)
  - Memory is composed of fixed-size addressable units
  - One instruction executes at a time, and is completed before the next instruction executes
- The model is not literally true, e.g., memory is hierarchical but made to “look flat”, doesn’t account for hardware and compiler optimizations

C directly implements this model in a HLL

Why Use Model That's Not Literally True?

- Simple is better, and many things--registers, floating point format--don't matter at all
- Avoid embedding assumptions where things could change …
  - Flat memory, though originally true, is no longer right, but we don't retrofit the model; we don't want people programming to a specific cache architecture, because it will likely change
    - Yes, exploit spatial locality
    - No, avoid blocking to fit in cache line, or tricking cache into prefetch, etc.
  - Compilers can add specific architectural optimizations (e.g., cache line size).
    - Portability via simple recompilation

Von Neumann Summary

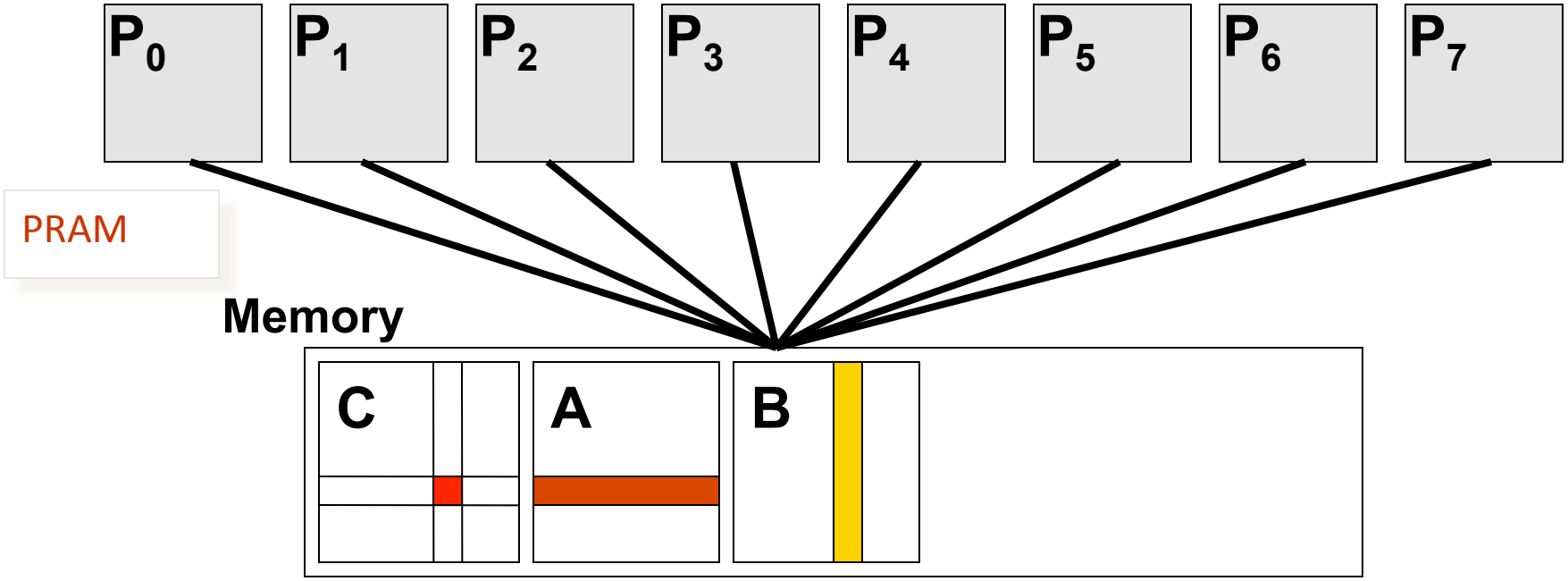
- The von Neumann model "explains" the costs of C because C expresses the facilities of the von Neumann machines in programming terms
- Knowing the relationship between C and the von Neumann machine is essential for writing fast programs
- Because so much code written to this model, HW vendors attempt to stay reasonably close
- These ideas are "in our bones" … it's how we think

What is the parallel version of von Neumann?

Recall Parallel Random Access Machine (PRAM) Model

PRAM has any number of processors

- Any memory can be referenced in "unit time"
- Memory read/write collisions must be resolved

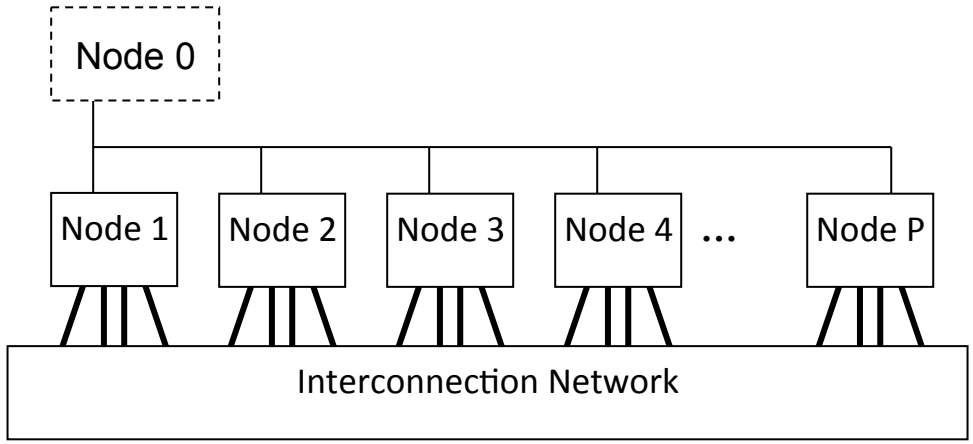


PRAM Often Proposed As A Candidate

- PRAM (Parallel RAM) ignores memory organization, collisions, latency, conflicts, etc.
- Ignoring these are *claimed* to have benefits ...
  - Portable everywhere since it is very general
  - It is a simple programming model ignoring only insignificant details – e.g., "only off by log P"
  - Ignoring memory difficulties is OK because hardware can "fake" a shared memory
  - Good for getting started: Begin with PRAM then refine the program to a practical solution if needed
- But locality very important to performance in *most* parallel architectures…

CTA Model

- Your text presents Candidate Type Architecture model:
  - *P* compute processors
  - 1 management processor
  - *d* degree (per-node connections to network)
  - 1 unit local memory latency
  - λ >> 1 global memory latency
- Node == processor + memory + NIC



What CTA Doesn't Describe

- CTA has no global memory … but memory can be globally *addressed*
- Mechanism for referencing global memory not specified: shared addressing, message passing, one-sided communication, …
- Interconnection network not specified
  - Does assume network, not bus (1 message at a time).
- λ is not specified beyond $\lambda >> 1$ -- cannot be because every machine is different

Communication Mechanisms

- Shared addressing
  - One consistent memory image – any node can load/store anywhere
  - Must protect locations from races
  - Some consider most convenient, but many challenges (performance/correctness)
  - CTA implies that best practice is to keep as much of the problem private; use sharing only to communicate – this style often encourages the opposite

Communication Mechanisms

- Message Passing
  - No global memory image; primitives are `send()` and `recv()`
  - Common in clusters and supercomputers
  - User writes in sequential language with message passing library:
    - Message Passing Interface (MPI) most common
  - Many people dislike, but it has been the dominant paradigm in HPC for a long time
    - The model forces you to think locally
    - Lots of high-performing legacy code => not likely to go away any time soon

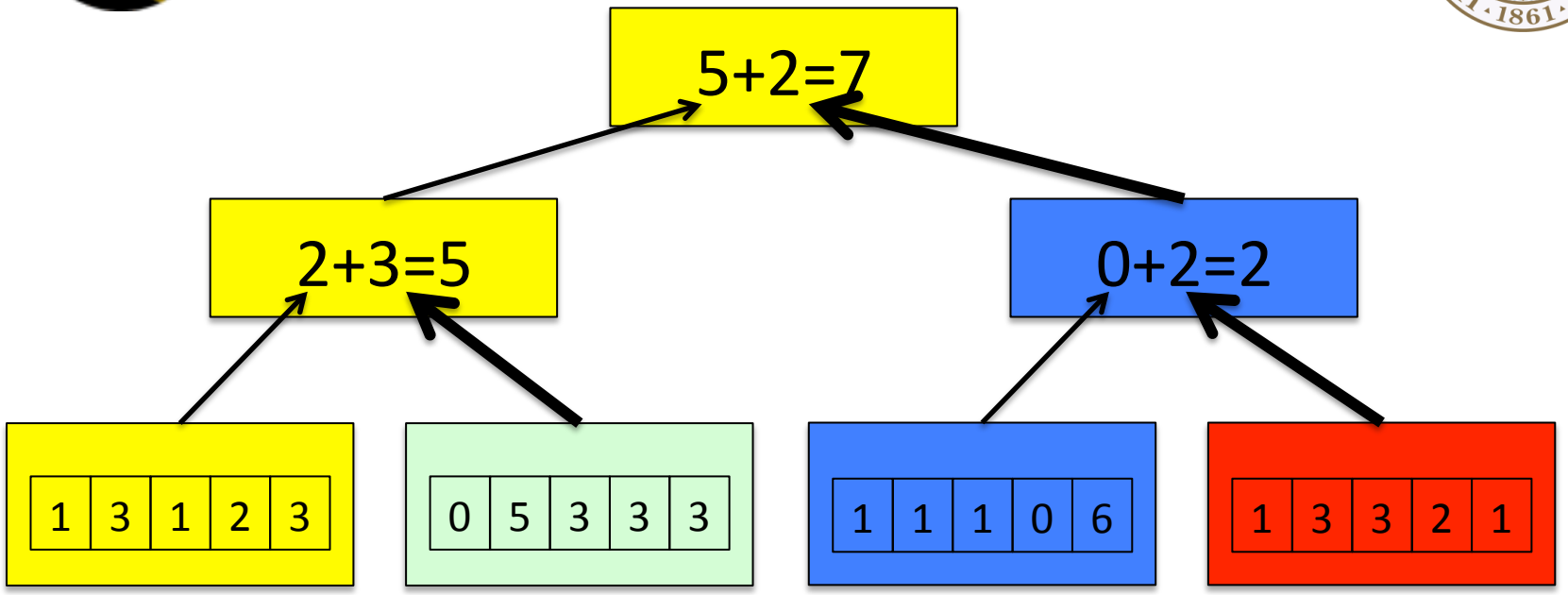
Communication Mechanisms

- One Sided Communication
  - One global address space; primitives are `get()` (fetch from remote) and `put()` (write to remote)
  - Many high-performing interconnects support RDMA: Remote Direct Memory Access (one-sided communication without involving other processor)
    - E.g., Infiniband, Cray Aries
    - Often very efficient – no remote involvement = low overhead
  - Consistency is the programmer's responsibility
  - Explicitly distinguishes local and remote
  - Either via library, or language-level support (Coarray Fortran and C++, UPC)

Apply CTA to Count 3s

- How does CTA guide us for Count 3s problem?
  - Assume small degree d (i.e., multiple communications to/from a node may cause congestion)
- What is the running time?

One solution



- Each processor computes 3's in its local chunk
- Combine via reduction tree – log P steps
- Cost $N/P + \lambda \log P$

Parallel Machine Model Summary

- Parallel hardware is a critical component of improving performance … but there's a Catch-22
  - To have portable programs, we must abstract away from the hardware
  - To write performant programs requires that we respect the hardware realities
- PRAM: Interesting theoretical model, but misses important details of most machines
- CTA: an abstract machine with just enough detail to support critical programming decisions
  - Highlights the importance of locality
- In homework you'll read about LogP model

Let's Revisit Memory

Memory Consistency Model: Rules that define how distinct tasks may view concurrent updates to memory

Strict Consistency

- All reads/writes to memory appear to happen at the same time on every thread/process/node
  - Intuitively, exactly what you would like
  - By definition, different tasks couldn't have simultaneous contradictory notions of memory
  - Requires some notion of globally consistent time to make any sense
  - And requires locking every non-private access
  - Not realistic, and really don't have any need for this level strictness
  - What do we really want?

What do we really want? (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

Consider this example – what possible outcomes would you intuitively expect?

What do we really want? (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

reg1 = 0, reg2 = 0

What do we really want? (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

reg1 = 0, reg2 = 0

What do we really want? (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

reg1 = 0, reg2 = 1

reg1 = 0, reg2 = 0

What do we really want? (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

reg1 = 2, reg2 = 0

reg1 = 0, reg2 = 1

reg1 = 0, reg2 = 0

What do we really want? (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

reg1 = 2, reg2 = 0

reg1 = 0, reg2 = 1

reg1 = 0, reg2 = 0

Really just want something that enforces one of these “intuitive outcomes”

A Slightly Weaker Model: Sequential Consistency

- Two parts to the definition:
  - All memory ops within a thread complete in program order
  - Across tasks, memory ops are interleaved in a consistent total order (everyone sees same interleaving)
- Intuitively: "An interleaving of the tasks' memory operations if they were instantaneous"
- Not as "strict" as strict, but still provides outcomes we'd "intuitively expect"
- Unfortunately, still untenable in general
  - guaranteeing a consistent, total order on memory ops again implies too much overhead

Reality … (adapted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

Sadly, yes – this can occur within most languages/architectures

The Real World … (not MTV)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the compiler" explanation:

- Traditionally, a compiler looks at a single task at a time
  (Practically speaking, it can't consider all possible potentially concurrent tasks)
- To a compiler looking at code in isolation, nothing prevents reordering as follows:

Code Snippet 1
y = 1
reg1 = x

Code Snippet 2
x = 2
reg2 = y

(at which point, obvious execution interleavings can yield the reg1 = 2, reg2 = 1 result).

The Real World … (not MTV)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors don't really execute one instruction, pause for completion, execute next. Instructions interleaved in pipeline, non-dependent instructions can proceed while awaiting memory references, etc.
- Consider shared x and y, living on different nodes:

The Real World … (not MTV)

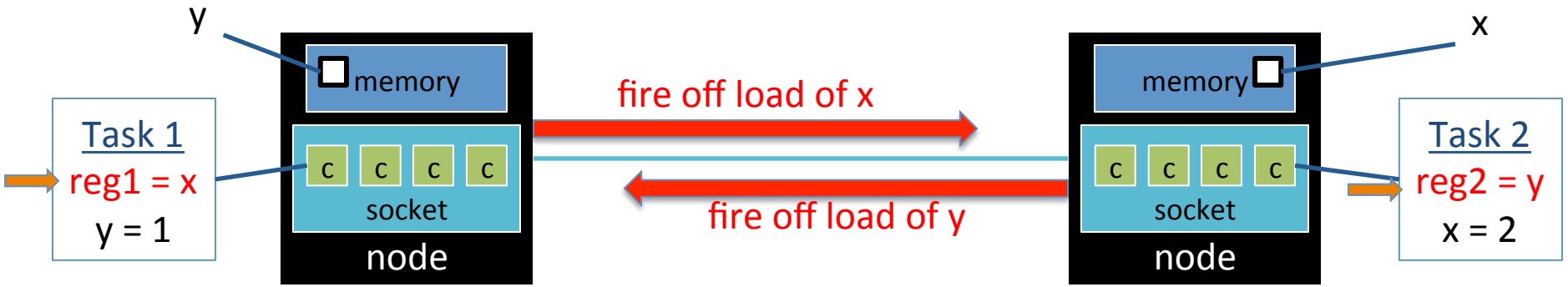
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors don't really execute one instruction, pause for completion, execute next. Instructions interleaved in pipeline, non-dependent instructions can proceed while awaiting memory references, etc.
- Consider shared x and y, living on different nodes:



The Real World … (not MTV)

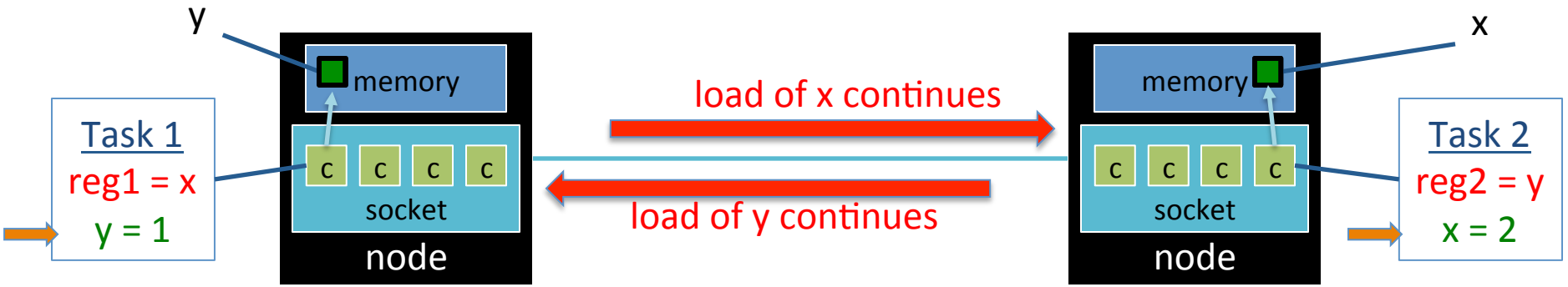
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors don't really execute one instruction, pause for completion, execute next. Instructions interleaved in pipeline, non-dependent instructions can proceed while awaiting memory references, etc.
- Consider shared x and y, living on different nodes:



The Real World … (not MTV)

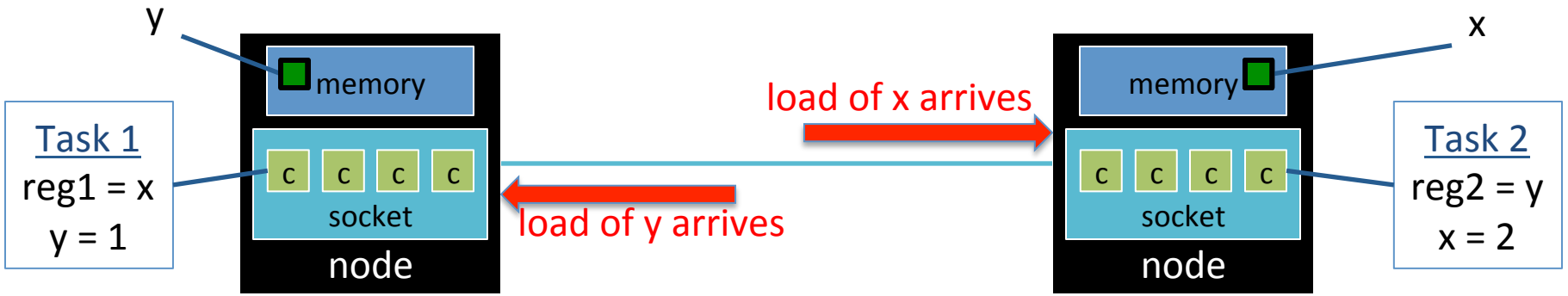
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors don't really execute one instruction, pause for completion, execute next. Instructions interleaved in pipeline, non-dependent instructions can proceed while awaiting memory references, etc.
- Consider shared x and y, living on different nodes:



The Real World … (not MTV)

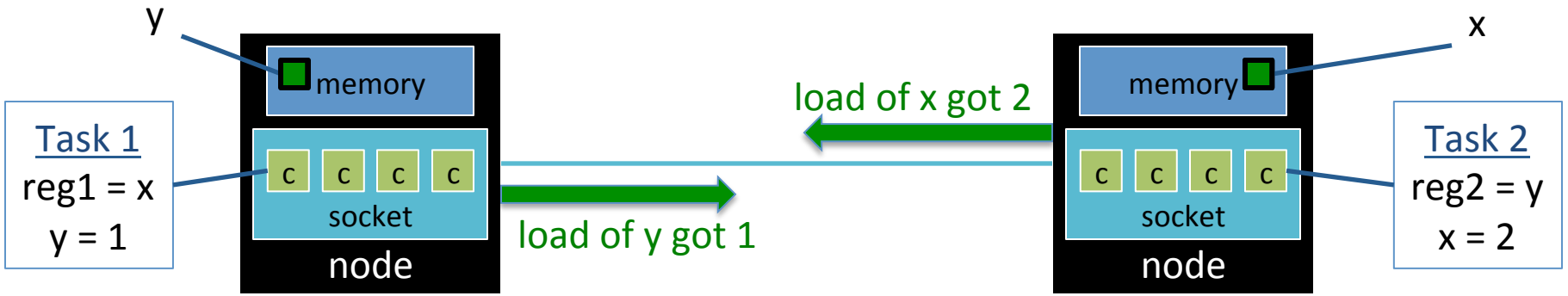
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x y = 1	reg2 = y x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors don't really execute one instruction, pause for completion, execute next. Instructions interleaved in pipeline, non-dependent instructions can proceed while awaiting memory references, etc.
- Consider shared x and y, living on different nodes:



Relaxed/Weak Consistency Models

- In practice, we generally have to deal with weaker consistency models
- Effort has gone into defining required Memory Consistency Models in language standards
- One common approach is "sequential consistency, provided you follow certain rules"
  - The rules mean the compiler and hardware can still optimize without fear of violating MCM
  - Java does this
  - C++11 introduced this

Discussion

- Some possible questions:
  - Are we really approaching the end of the multicore revolution?
  - What comes next? How can we continue to increase processor performance?
    - Or do we care? Should we concentrate on something else? Is the processor always or even often the bottleneck in your codes?
  - If we can't have all transistors on at once, how should we use them? Should this change the tradeoffs in designing hardware?