EECS 570

Lecture 2

Parallel Prog. Models

Fall 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Intel Paragon XP/S

Slides developed in part by Drs. Adve, Falsafi, Martin, Musuvathi, Narayanasamy, Nowatzyk, Wenisch, Sarkar, Mikko Lipasti, Jim Smith, John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström, and probably others

Announcements

This week Friday:

Will discuss programming assignment 1

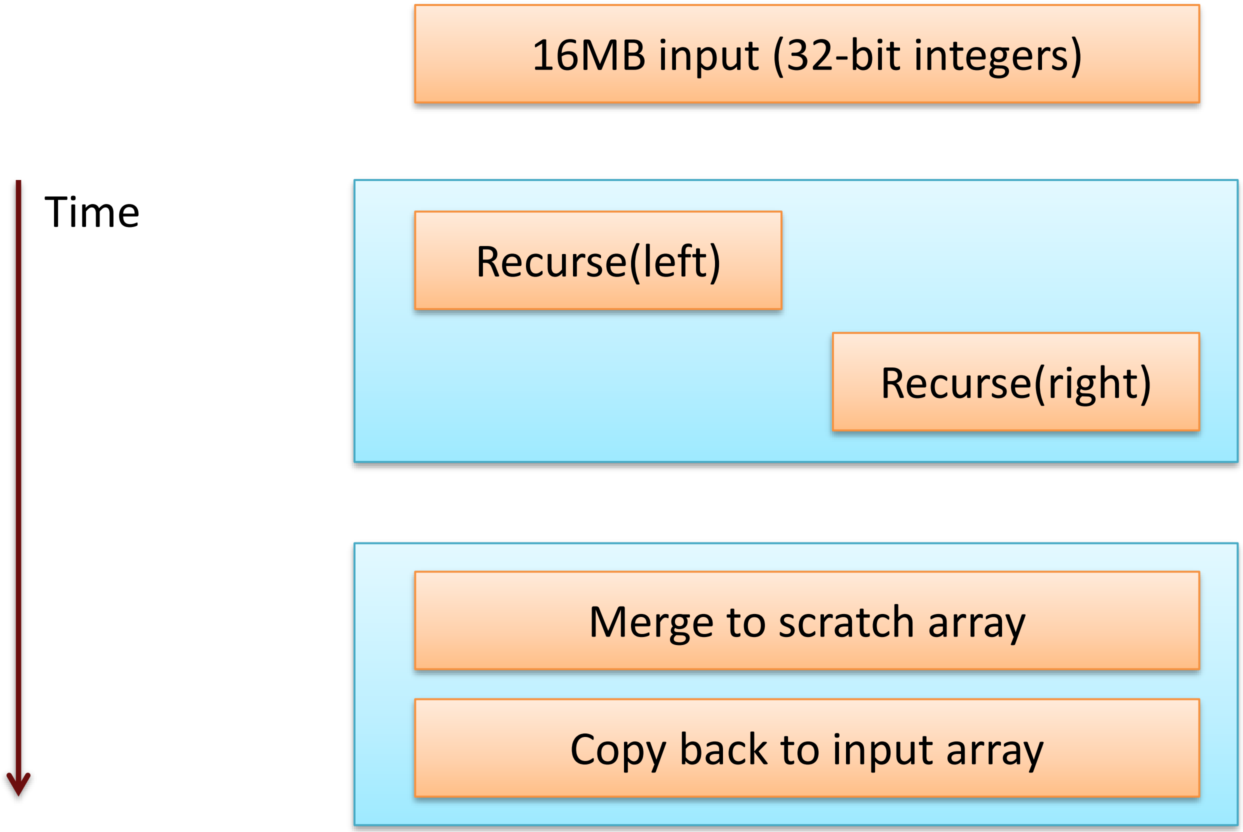
Readings

For Monday

- David Wood and Mark Hill. “Cost-Effective Parallel Computing,” *IEEE Computer*, 1995.
- Mark Hill et al. “21st Century Computer Architecture.” CCC White Paper, 2012.

Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. Commun. ACM 61, July 2018

Sequential Merge Sort



16MB input (32-bit integers)

Time

Recurse(left)

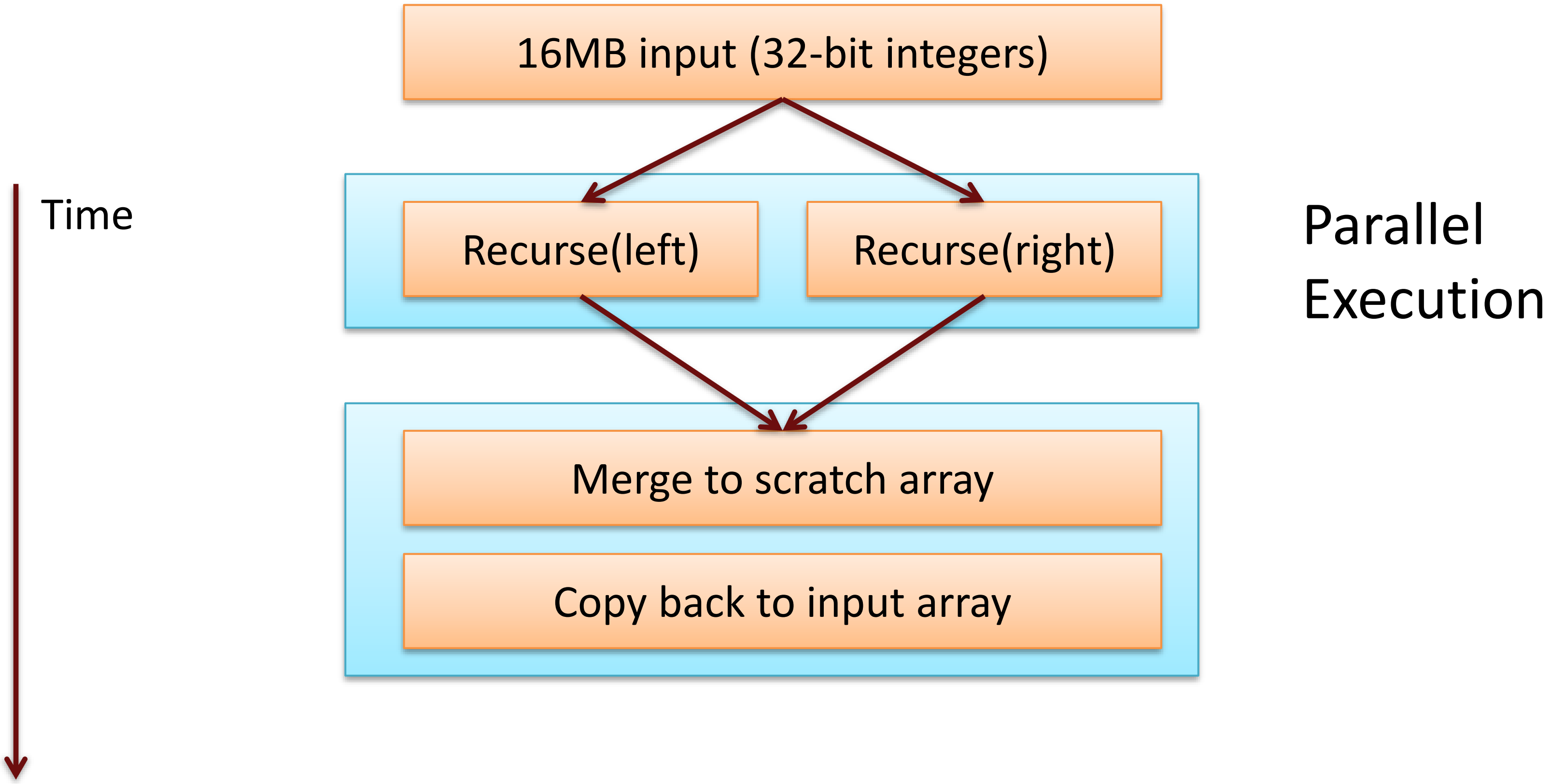
Recurse(right)

Merge to scratch array

Copy back to input array

Sequential
Execution

Parallel Merge Sort (as Parallel Directed Acyclic Graph)



Time

16MB input (32-bit integers)

Recurse(left)

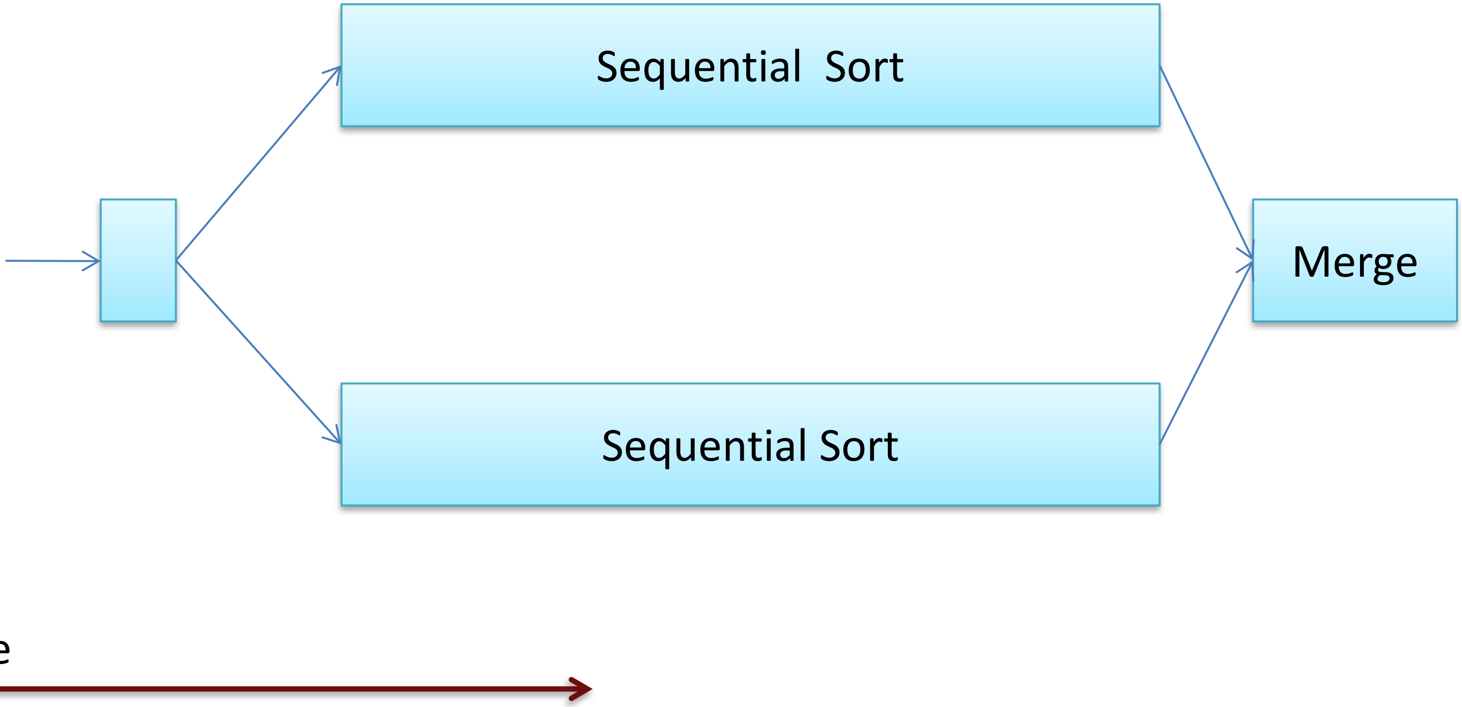
Recurse(right)

Parallel Execution

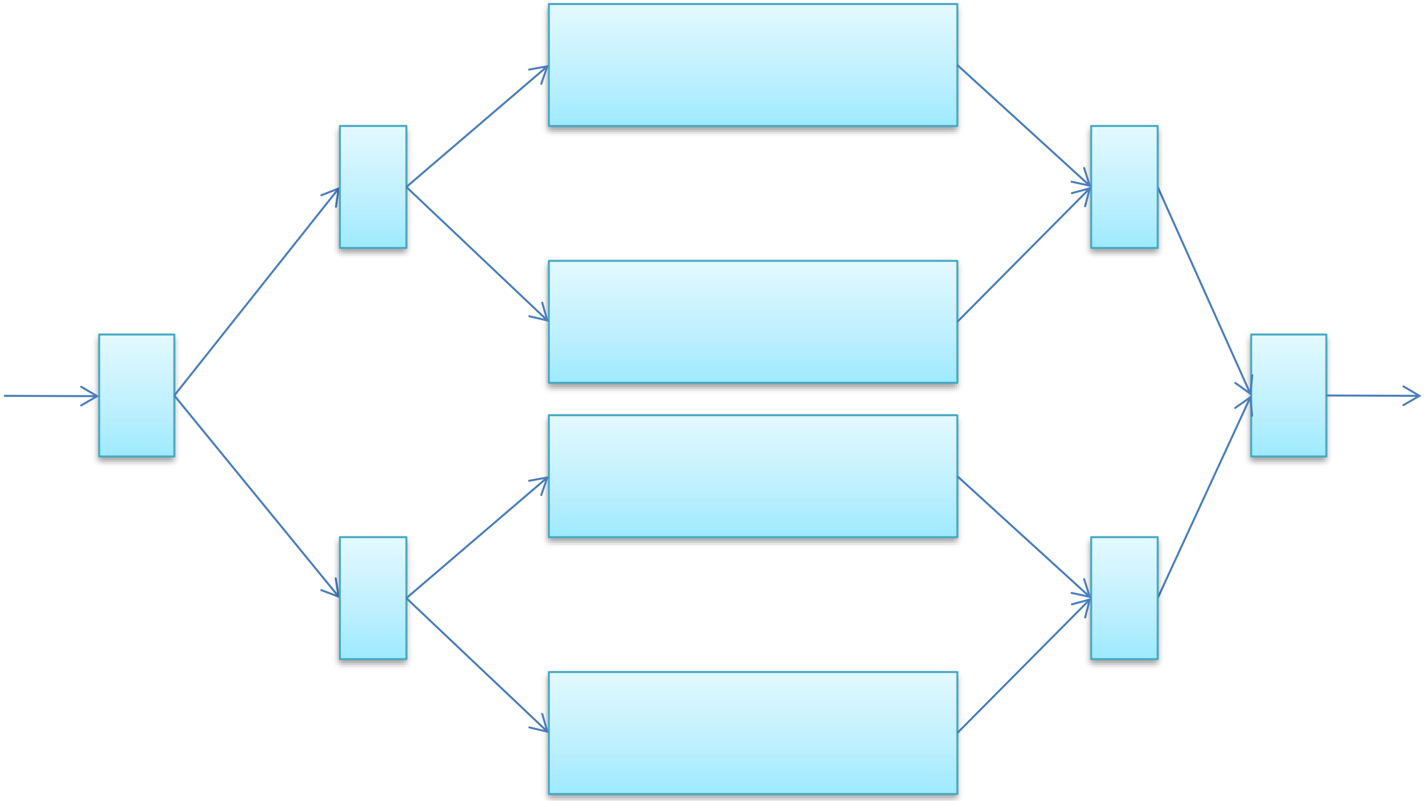
Merge to scratch array

Copy back to input array

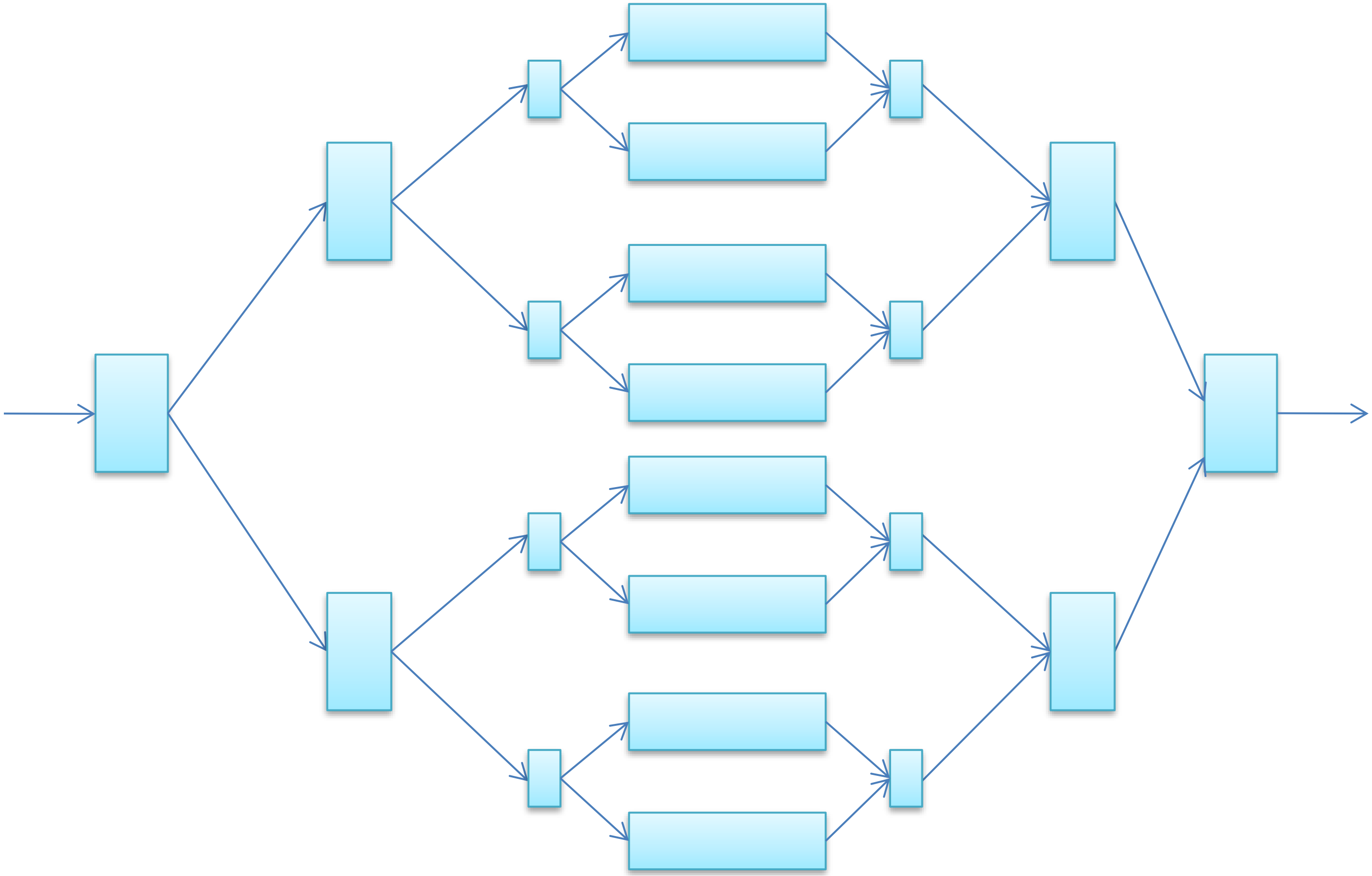
Parallel DAG for Merge Sort (2-core)



Parallel DAG for Merge Sort (4-core)



Parallel DAG for Merge Sort (8-core)

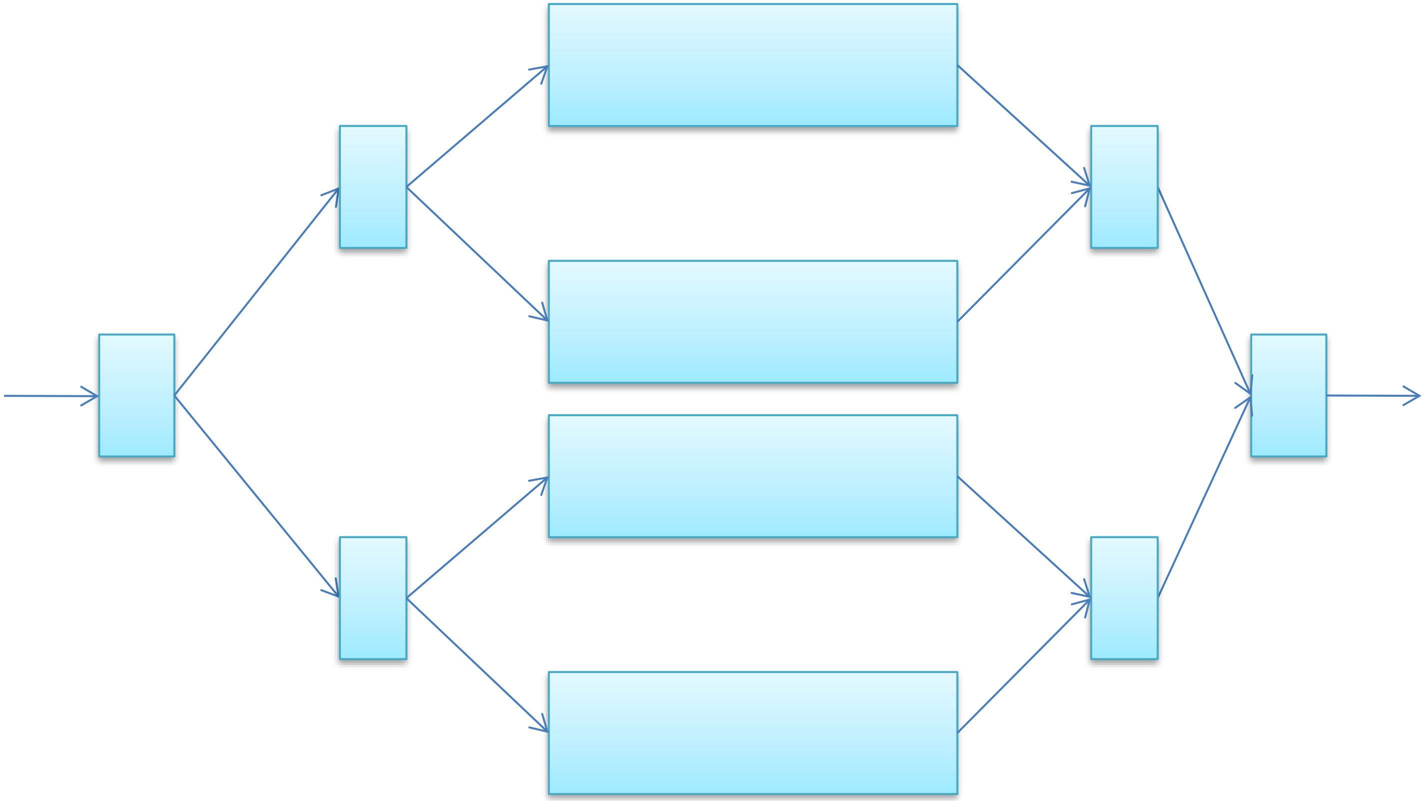


The DAG Execution Model of a Parallel Computation

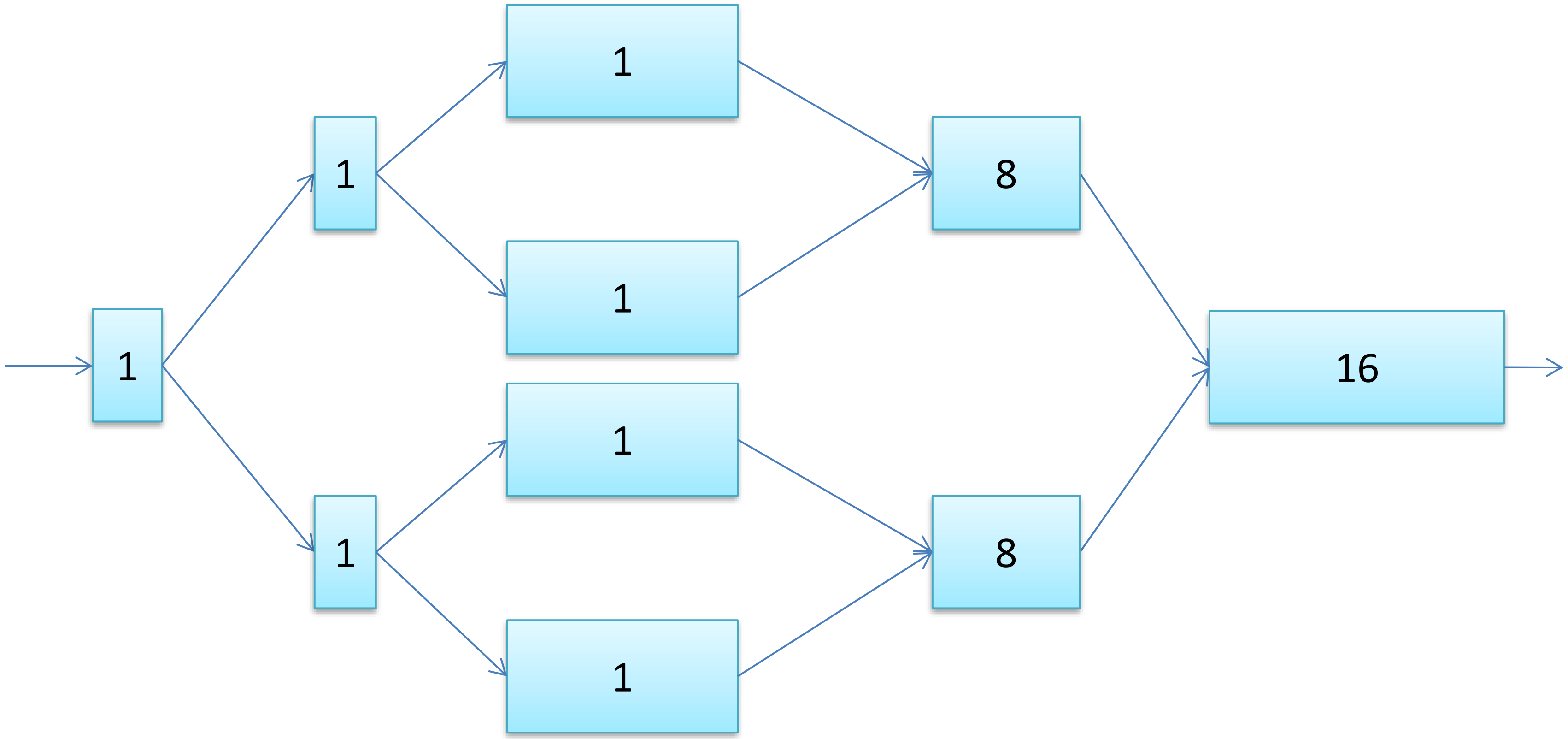
- Given an input, dynamically create a DAG

- Nodes represent sequential computation
  - ❒ Weighted by the amount of work
- Edges represent dependencies:
  - ❒ Node A → Node B means that B cannot be scheduled unless A is finished

Sorting 16 elements in four cores



Sorting 16 elements in four cores (4 element arrays sorted in constant time)



Performance Measures

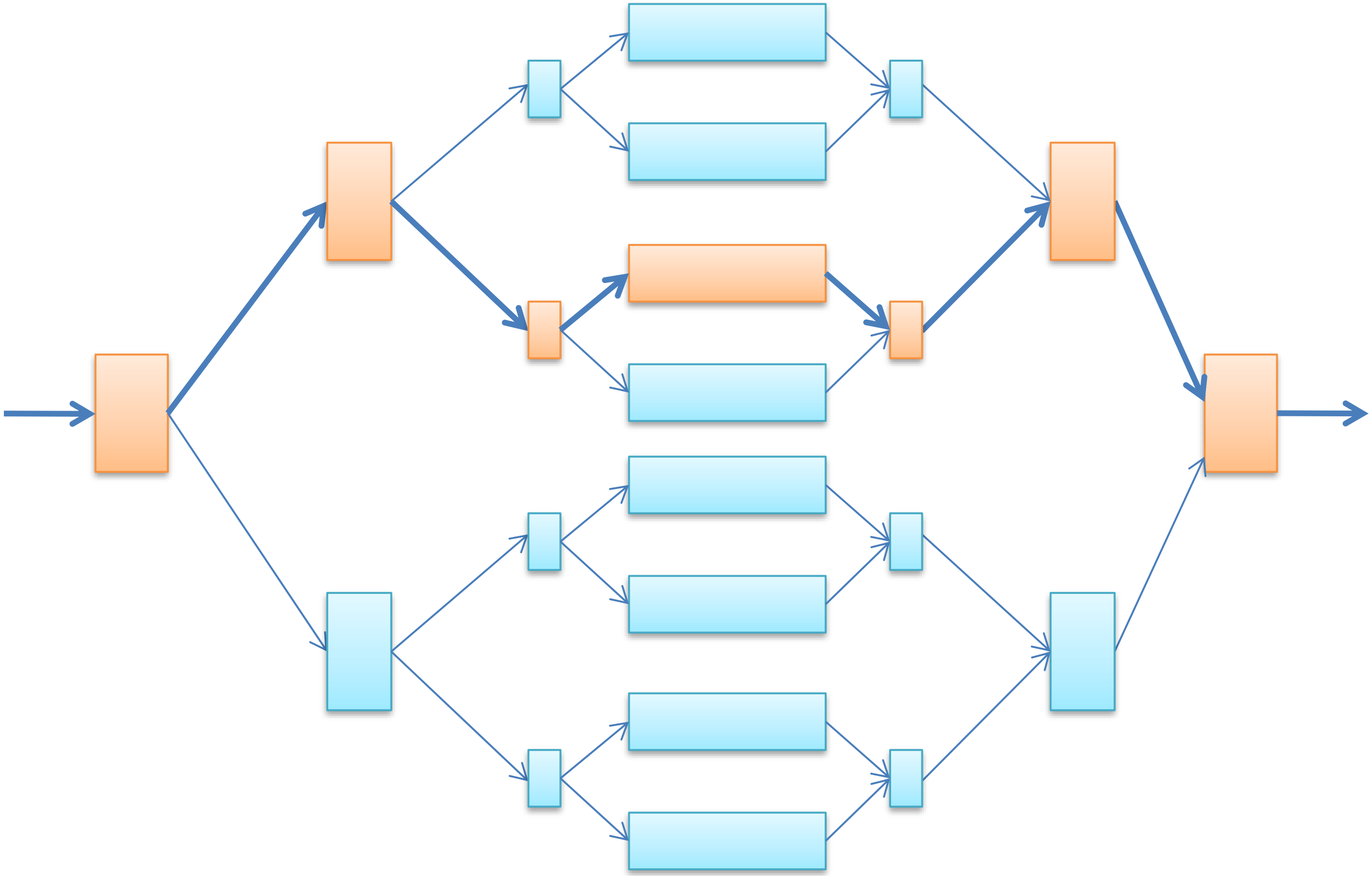
- Given a graph G, a scheduler S, and P processors
- $T_p(S)$: Time on P processors using scheduler S
- T_p : Time on P processors using best scheduler
- T_1 : Time on a single processor (sequential cost)
- T_∞ : Time assuming infinite resources

Work and Depth

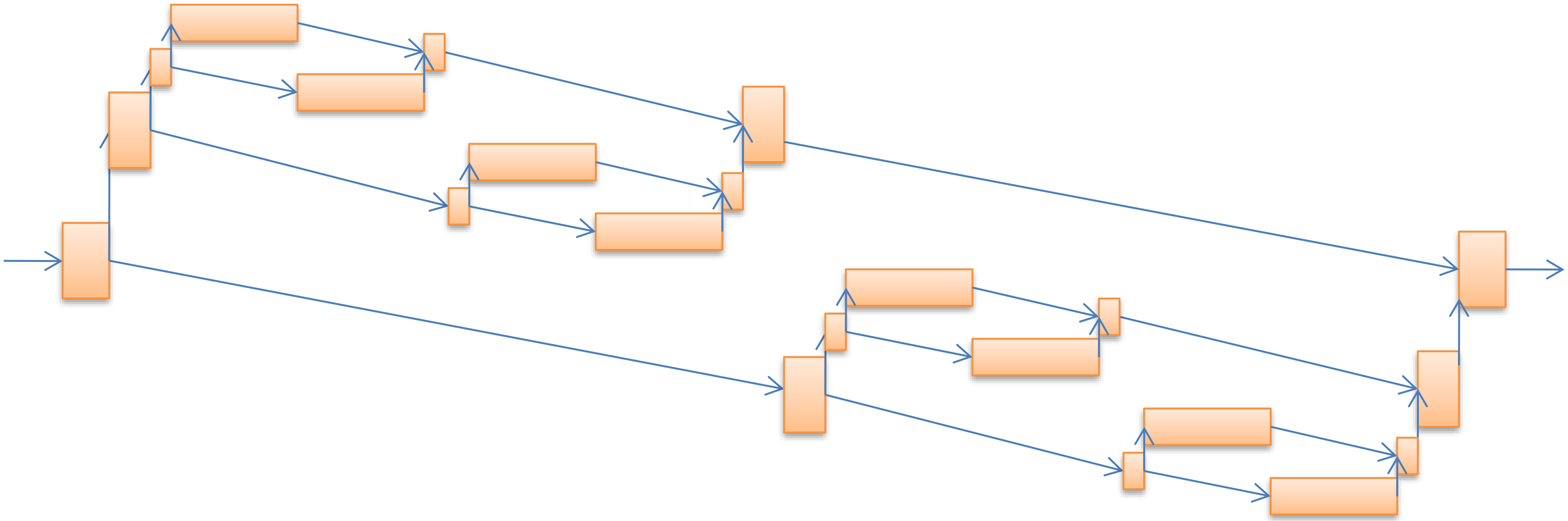
- T_1 = Work
  - ❒ The total number of operations executed by a computation

- T_∞ = Depth
  - ❒ The longest chain of sequential dependencies (critical path) in the parallel DAG

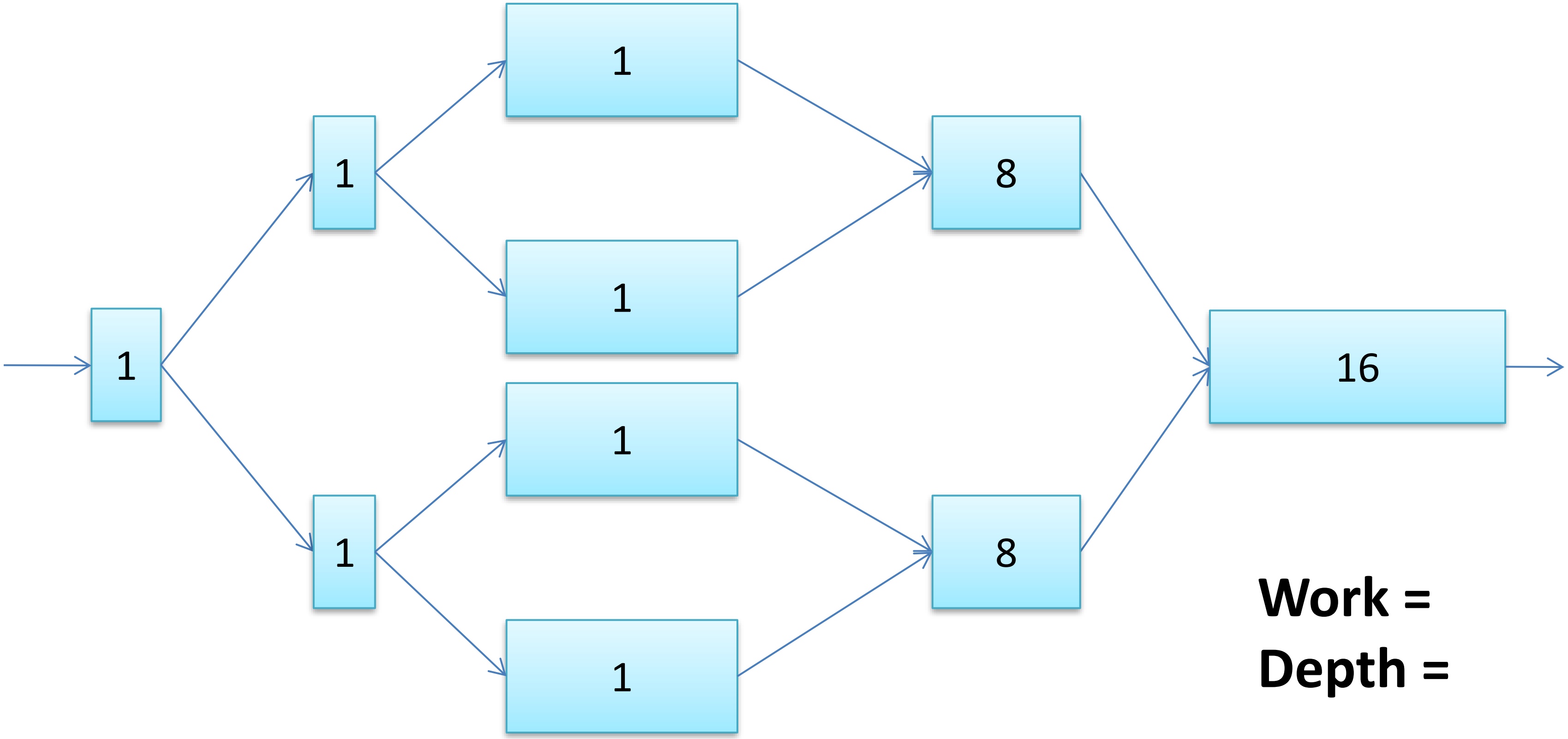
T_∞ (Depth): Critical Path Length (Sequential Bottleneck)



T_1 (work): Time to Run Sequentially



Sorting 16 elements in four cores (4 element arrays sorted in constant time)



Work =

Depth =

Some Useful Theorems

Work Law

- “You cannot avoid work by parallelizing”

$$T_1 / P \leq T_P$$

Work Law

- "You cannot avoid work by parallelizing"

$$T_1 / P \leq T_P$$

$$\text{Speedup} = T_1 / T_P$$

Work Law

- “You cannot avoid work by parallelizing”

$$T_1 / P \leq T_P$$

$$\text{Speedup} = T_1 / T_P$$

- Can speedup be more than 2 when we go from 1-core to 2-core in practice?

Depth Law

- More resources should make things faster
- You are limited by the sequential bottleneck

$$T_P \geq T_\infty$$

Amount of Parallelism

Parallelism = T_1 / T_∞

Maximum Speedup Possible

Speedup $T_1 / T_P \leq T_1 / T_\infty$ **Parallelism**

"speedup is bounded above
by available parallelism"

Greedy Scheduler

- If more than P nodes can be scheduled, pick any subset of size P

- If less than P nodes can be scheduled, schedule them all

Performance of the Greedy Scheduler

$$T_P(\text{Greedy}) \leq T_1 / P + T_\infty$$

Work law $T_1 / P \leq T_P$

Depth law $T_\infty \leq T_P$

Greedy is optimal within factor of 2

$T_P \leq T_P(\text{Greedy}) \leq 2\ T_P$

Work law $T_1 / P \leq T_P$

Depth law $T_\infty \leq T_P$

Work/Depth of Merge Sort (Sequential Merge)

- Work T_1 : $O(n \log n)$
- Depth T_∞ : $O(n)$
  - ❐ Takes $O(n)$ time to merge n elements
- Parallelism:
  - ❐ $T_1 / T_\infty = O(\log n)$ → really bad!

Main Message

- Analyze the Work and Depth of your algorithm
- Parallelism is Work/Depth
- Try to decrease Depth
  - ❐ the critical path
  - ❐ a *sequential* bottleneck
- If you increase Depth
  - ❐ better increase Work by a lot more!

Amdahl's law

- Sorting takes 70% of the execution time of a sequential program
- You replace the sorting algorithm with one that scales perfectly on multi-core hardware
- How many cores do you need to get a 4x speed-up on the program?

Amdahl's law, $f = 70\%$

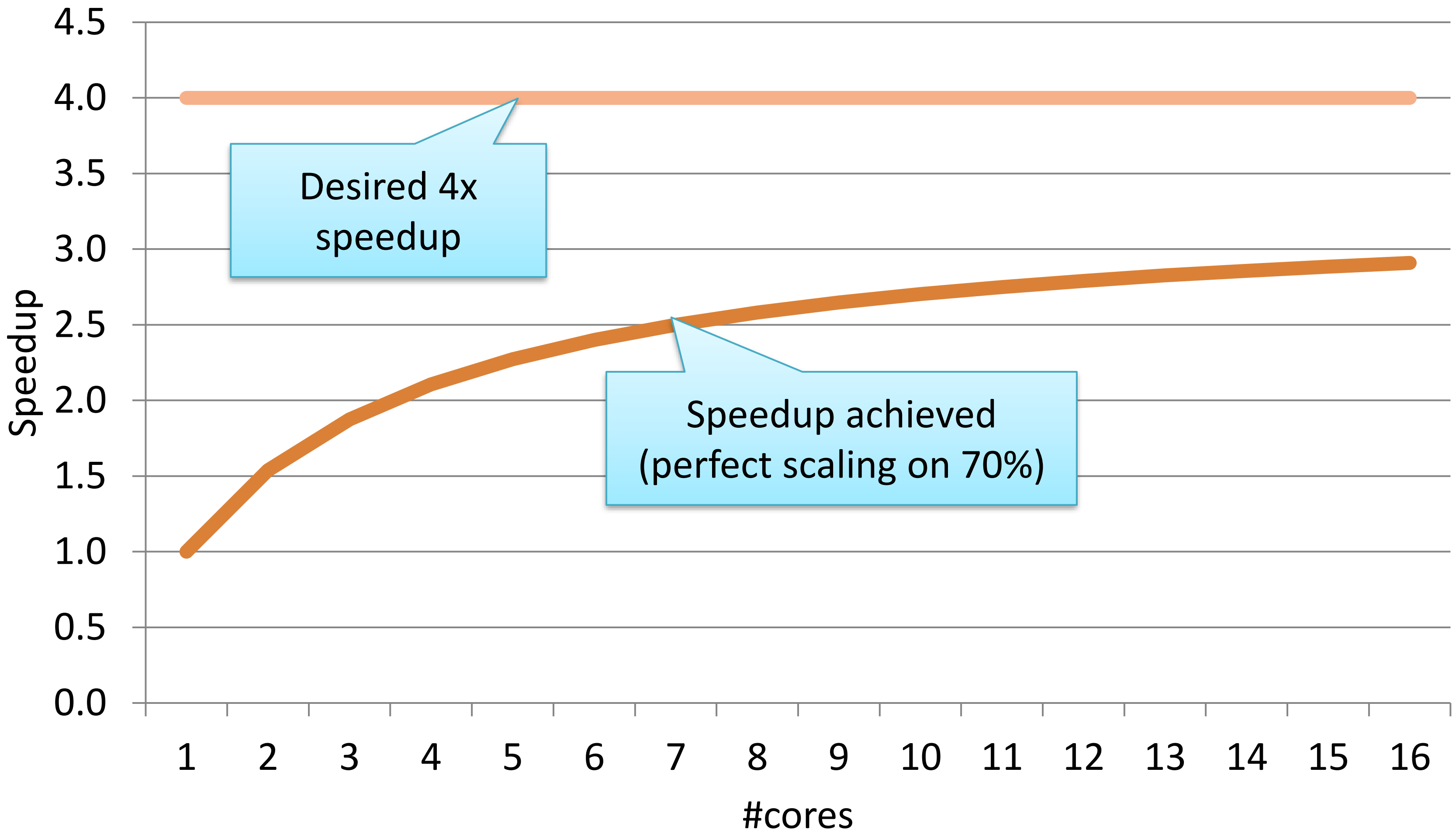
Speedup(f, c) $= 1 / (1 - f) + f / c$

f = the parallel portion of execution

$1 - f$ = the sequential portion of execution

c = number of cores used

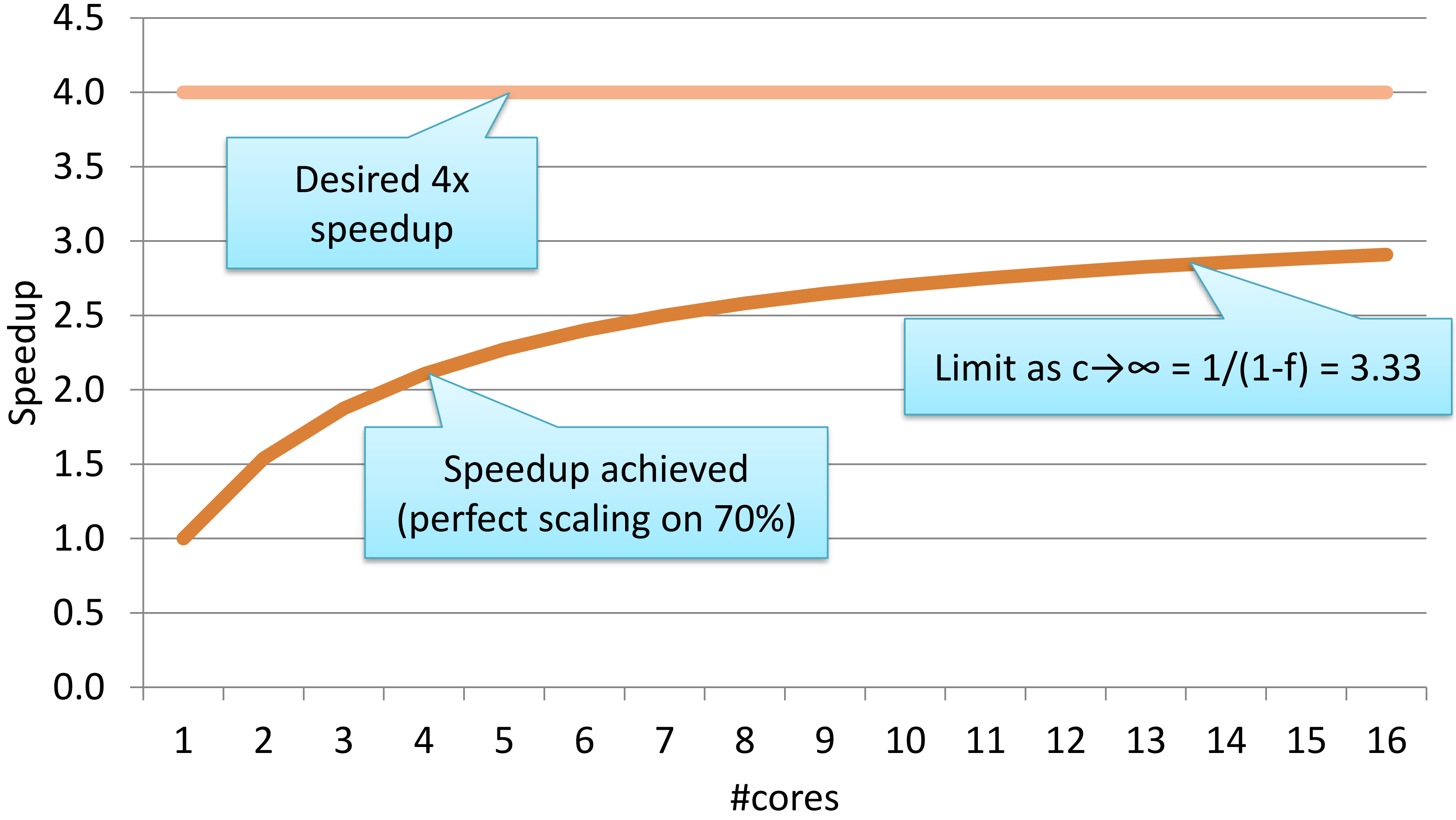
Amdahl's law, $f = 70\%$



Desired 4x speedup

Speedup achieved (perfect scaling on 70%)

Amdahl's law, $f = 70\%$

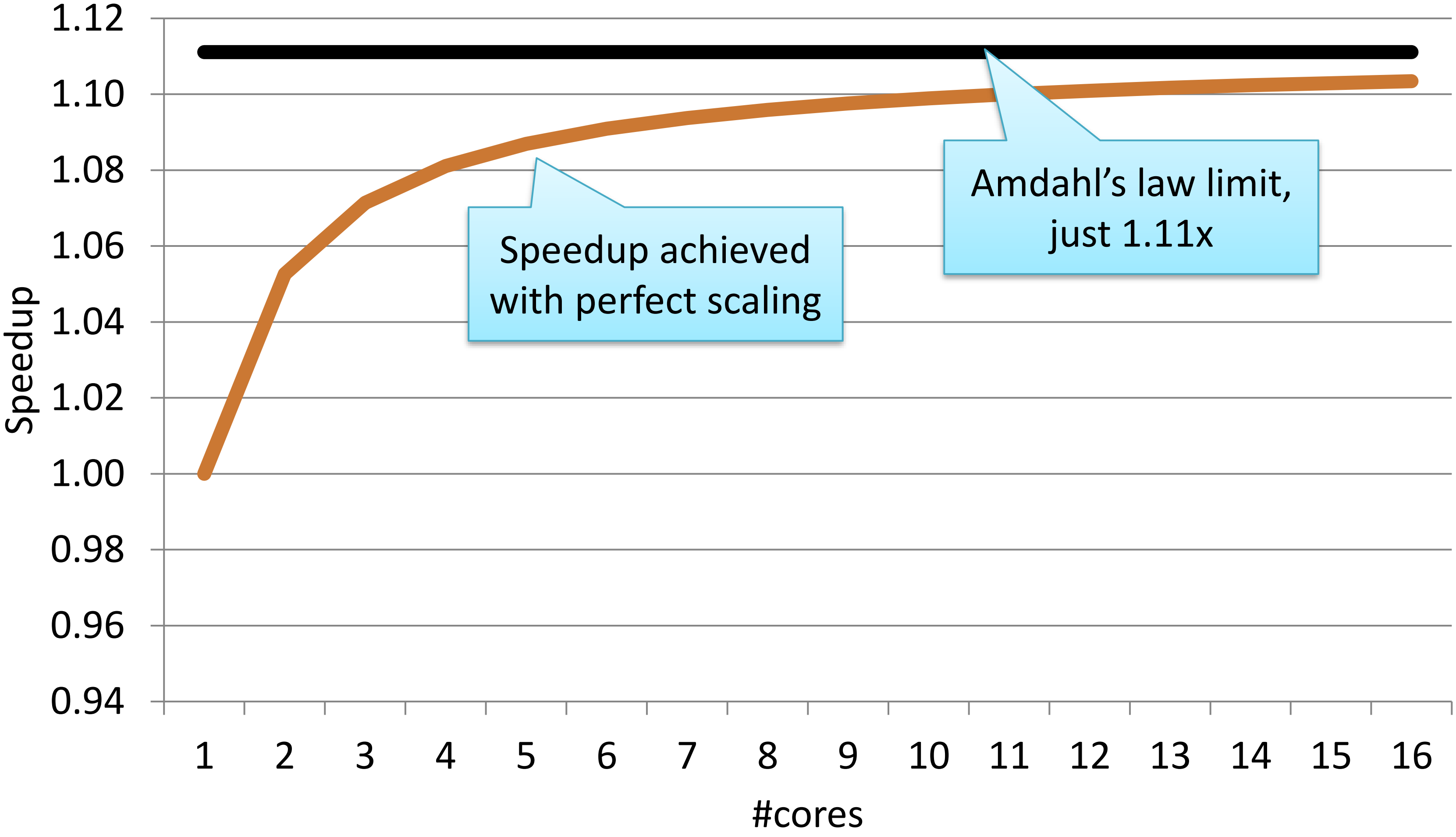


Desired 4x speedup

Limit as c→∞ = 1/(1-f) = 3.33

Speedup achieved (perfect scaling on 70%)

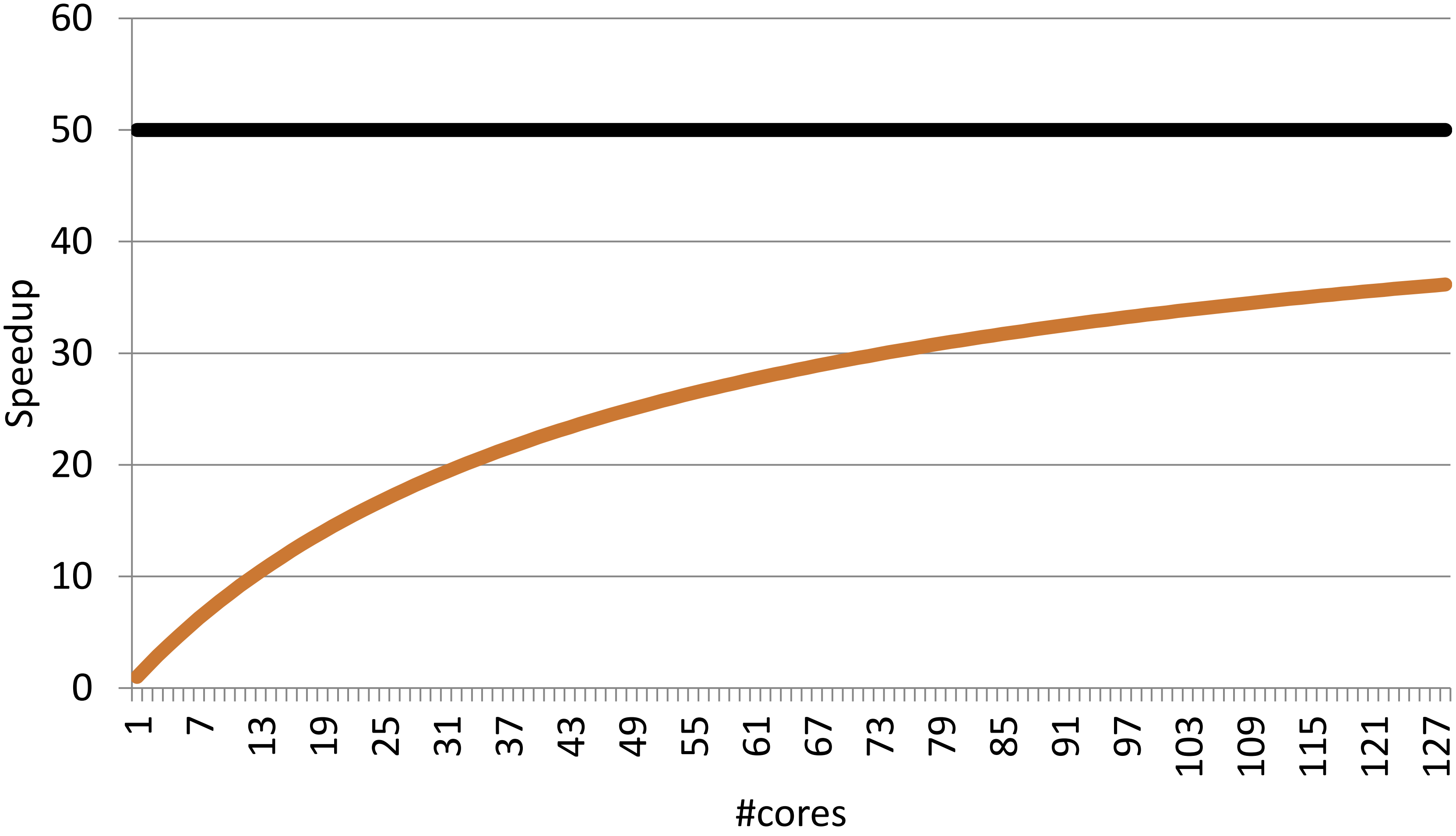
Amdahl's law, $f = 10\%$



Speedup achieved with perfect scaling

Amdahl's law limit, just 1.11x

Amdahl's law, $f = 98\%$



Lesson

- Speedup is limited by <u>sequential</u> code

- Even a small percentage of <u>sequential</u> code can greatly limit potential speedup

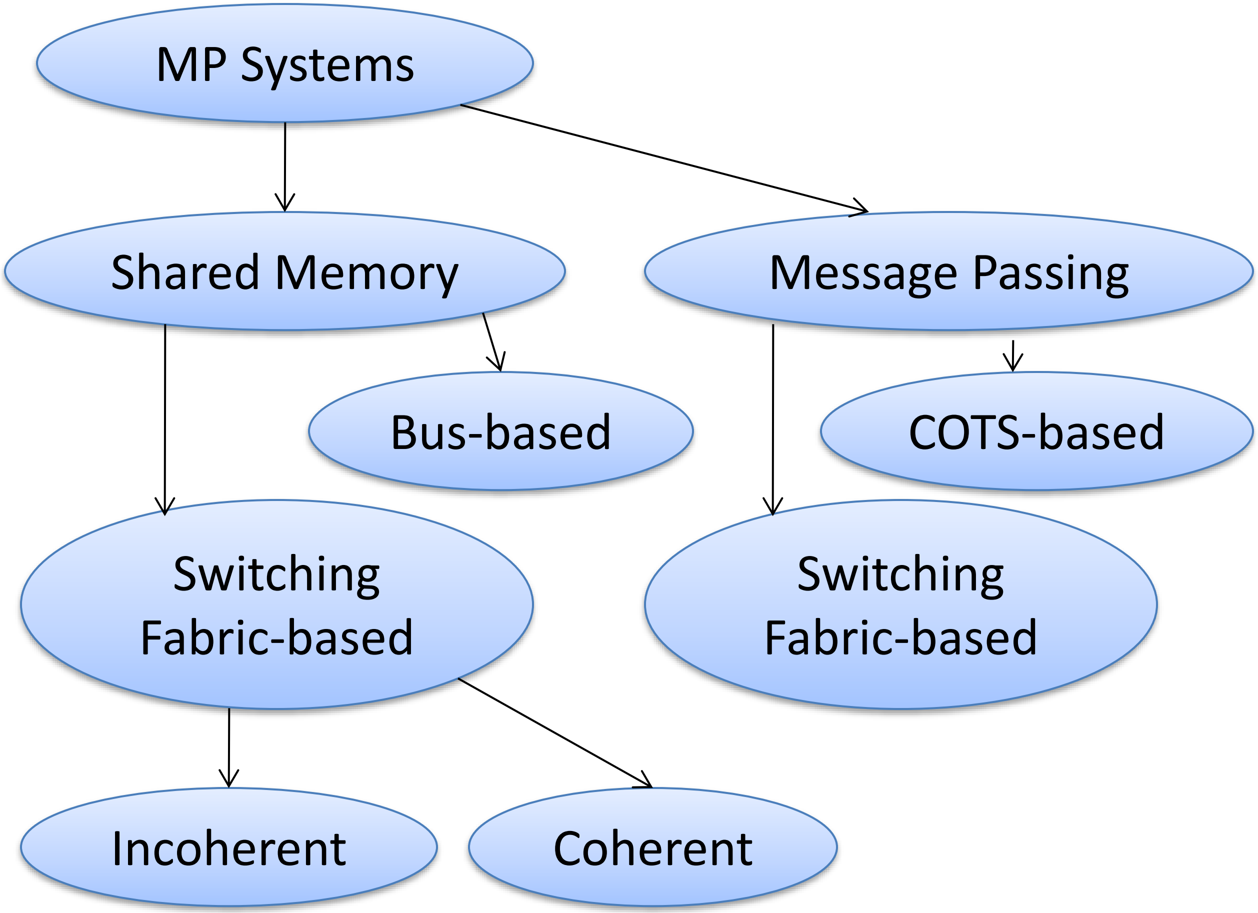
Parallel Programming Models and Interfaces

Programming Models

- High level paradigm for expressing an algorithm
  - ❒ Examples:
    - ❍ Functional
    - ❍ Sequential, procedural
    - ❍ Shared memory
    - ❍ Message Passing
- Embodied in languages that support concurrent execution
  - ❒ Incorporated into language constructs
  - ❒ Incorporated as libraries added to existing sequential language
- Top level features:

  (For conventional models – shared memory, message passing)
  - ❒ Multiple threads are conceptually visible to programmer
  - ❒ Communication/synchronization are visible to programmer
    - ❍ Somewhat implicit for shared memory

An Incomplete Taxonomy

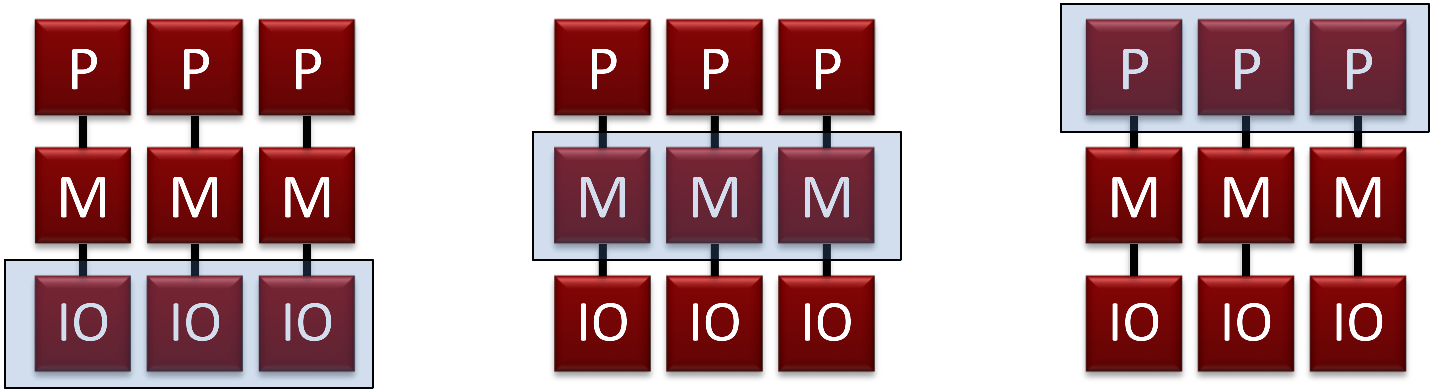


- VLIW
- SIMD
- Vector
- Data Flow
- GPU
- MapReduce / WSC
- Systolic Array
- Reconfigurable / FPGA
- …

Programming Model Elements

- For both Shared Memory and Message Passing
- Processes and threads
  - ❒ ***Process:*** A shared address space and one or more threads of control
  - ❒ ***Thread:*** A sequence of instructions
  - ❒ ***Task***: Less formal term – part of an overall job
  - ❒ Created, terminated, scheduled, etc.
- Communication
  - ❒ Passing of data
- Synchronization
  - ❒ Communicating control information
  - ❒ To assure reliable, correct communication

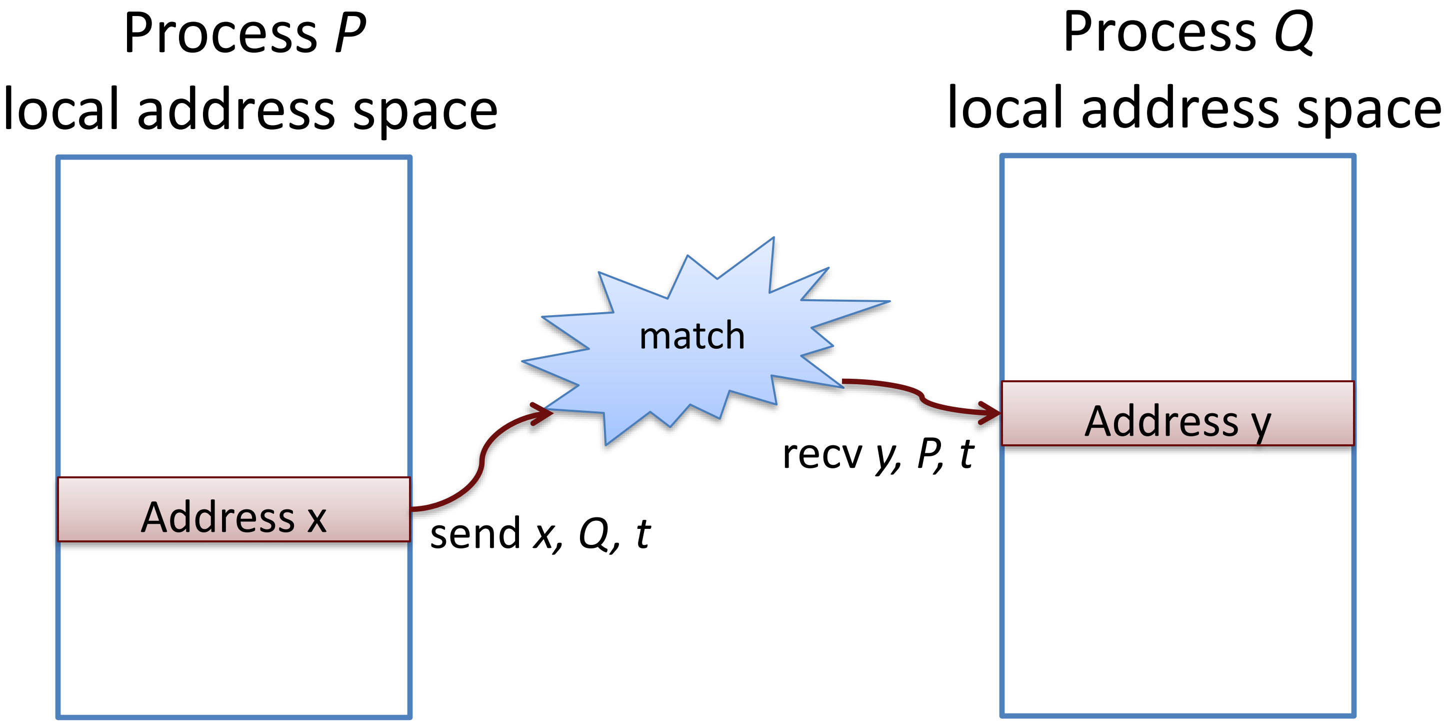
Historical View



Join at:	I/O (Network)	Memory	Processor
Program with:	Message passing	Shared Memory	Dataflow, SIMD, VLIW, CUDA, other data parallel

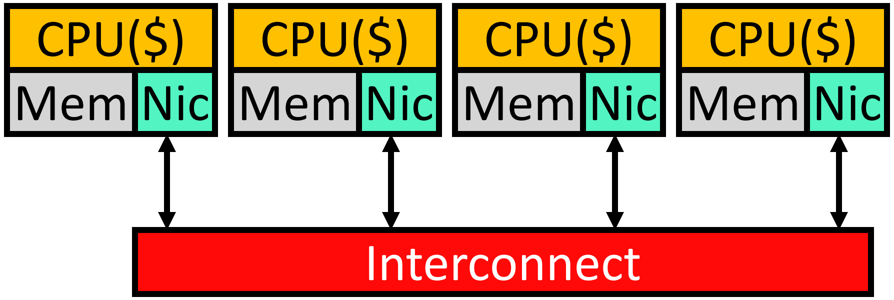
Message Passing Programming Model

Message Passing Programming Model



- User level send/receive abstraction
  - ❒ Match via local buffer (x,y), process (Q,P), and tag (t)
  - ❒ Need naming/synchronization conventions

Message Passing Architectures



- Cannot directly access memory of another node
- IBM SP-2, Intel Paragon, Myrinet Quadrics QSW
- Cluster of workstations (e.g., MPI on flux cluster)

MPI -Message Passing Interface API

- A widely used standard
  - ❒ For a variety of distributed memory systems
    - SMP Clusters, workstation clusters, MPPs, heterogeneous systems
- Also works on Shared Memory MPs
  - ❒ Easy to emulate distributed memory on shared memory HW
- Can be used with a number of high level languages
- Available in the Flux cluster at Michigan

Processes and Threads in Message Passing

- Common: multiple threads/processes with different address spaces
  - ❒ No shared memory
  - ❒ Communication has to be explicit through sending and receiving of messages
- Processes may also be running on different OSes
  - ❒ Process creation often external to execution environment; e.g. shell script
  - ❒ Hard for user process on one system to create process on another OS
- Lots of flexibility (advantage of message passing). Could have:
  1. Multiple threads sharing an address space
  2. Multiple processes sharing an address space
- 1 and 2 easily implemented on shared memory HW (with single OS)
  - ❒ Process and thread creation/management similar to shared memory

Communication and Synchronization

- Combined in the message passing paradigm
  - ❒Synchronization of messages part of communication semantics
- Point-to-point communication
  - ❒From one process to another
- Collective communication
  - ❒Involves groups of processes
  - ❒e.g., broadcast

Message Passing: Send()

- Send(<what>, <where-to>, <how>)
- What:
  - ❐ A data structure or object in user space
  - ❐ A buffer allocated from special memory
  - ❐ A word or signal
- Where-to:
  - ❐ A specific processor
  - ❐ A set of specific processors
  - ❐ A queue, dispatcher, scheduler
- How:
  - ❐ Asynchronously vs. synchronously
  - ❐ Typed
  - ❐ In-order vs. out-of-order
  - ❐ Prioritized

Message Passing: Receive()

- Receive(<data>, <info>, <what>, <how>)
- Data: mechanism to return message content
  - ❒ A buffer allocated in the user process
  - ❒ Memory allocated elsewhere
- Info: meta-info about the message
  - ❒ Sender-ID
  - ❒ Type, Size, Priority
  - ❒ Flow control information
- What: receive only certain messages
  - ❒ Sender-ID, Type, Priority
- How:
  - ❒ Blocking vs. non-blocking

Synchronous vs Asynchronous

- Synchronous Send
  - ❒Stall until message has actually been received
  - ❒Implies a message acknowledgement from receiver to sender
- Synchronous Receive
  - ❒Stall until message has actually been received
- Asynchronous Send and Receive
  - ❒Sender and receiver can proceed regardless
  - ❒Returns *request handle* that can be tested for message receipt
  - ❒Request handle can be tested to see if message has been sent/received

Deadlock

- Blocking communications may deadlock

```
<Process 0>
Send(Process1, Message);
Receive(Process1, Message);
```

```
<Process 1>
Send(Process0, Message);
Receive(Process0, Message);
```

- Requires careful (safe) ordering of sends/receives

```
<Process 0>
Send(Process1, Message);
Receive(Process1, Message);
```

```
<Process 1>
Receive (Process0, Message);
Send (Process0, Message);
```

Message Passing Paradigm Summary

Programming Model (Software) point of view:

- Disjoint, separate name spaces
- “Shared nothing”
- Communication via explicit, typed messages: send & receive

Message Passing Paradigm Summary

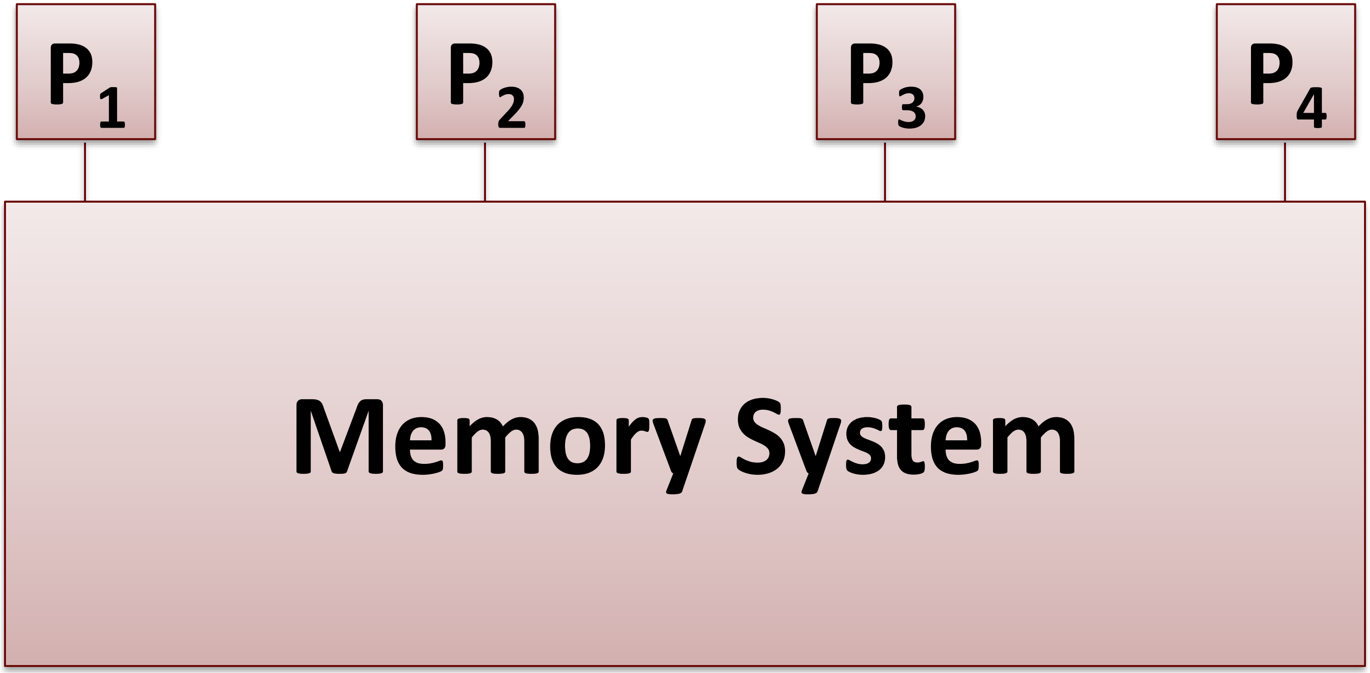
Computer Engineering (Hardware) point of view:

- Treat inter-process communication as I/O device
- Critical issues:
  - ❒ How to optimize API overhead
  - ❒ Minimize communication latency
  - ❒ Buffer management: how to deal with early/unsolicited messages, message typing, high-level flow control
  - ❒ Event signaling & synchronization
  - ❒ Library support for common functions (barrier synchronization, task distribution, scatter/gather, data structure maintenance)

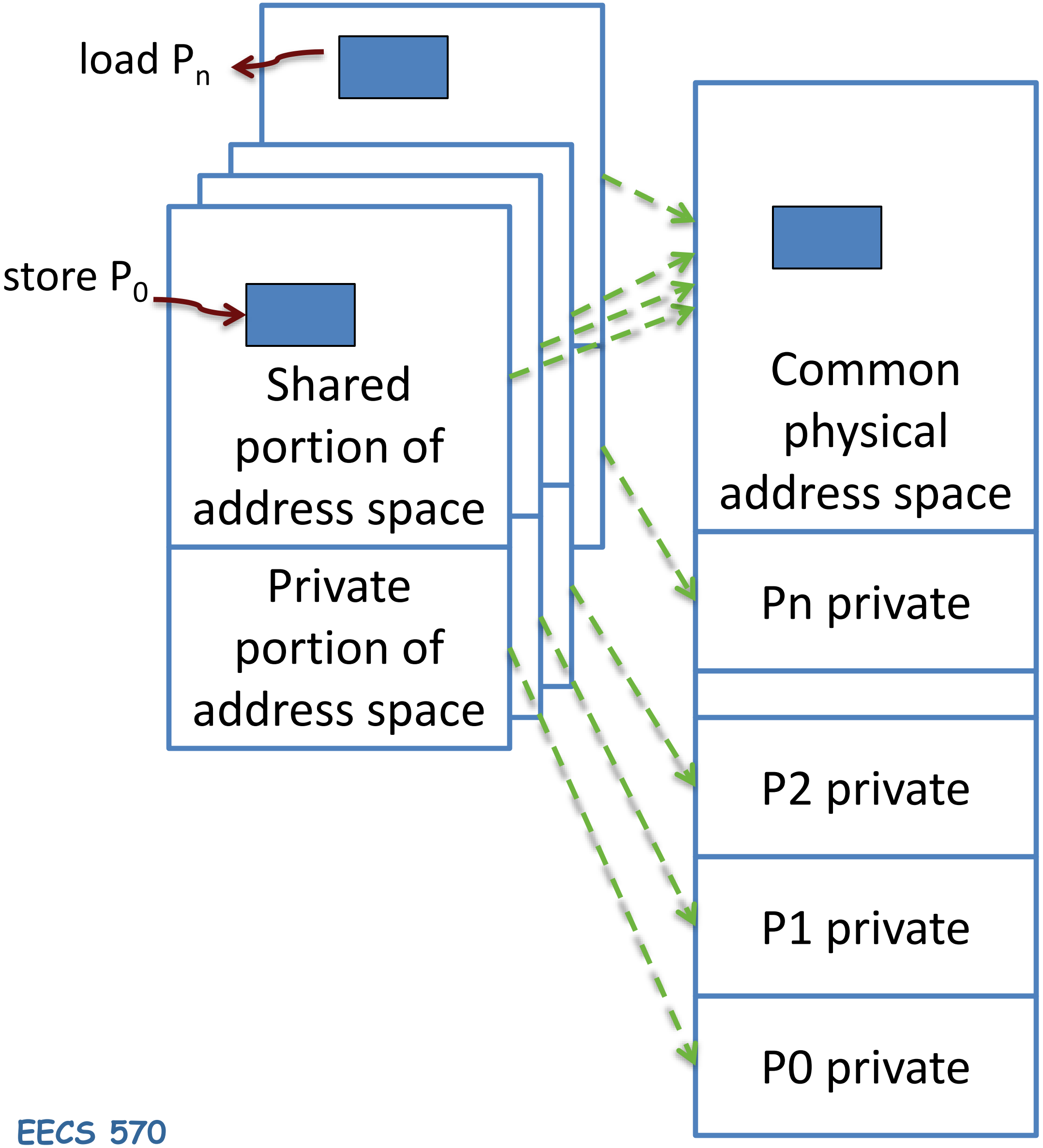
Shared Memory Programming Model

Shared-Memory Model

- Multiple execution contexts sharing a single address space
  - Multiple programs (MIMD)
  - Or more frequently: multiple copies of one program (SPMD)
- Implicit (automatic) communication via loads and stores
- Theoretical foundation: PRAM model



Global Shared Physical Address Space


load P_n
store P_0
Shared portion of address space
Private portion of address space
Common physical address space
Pn private
P2 private
P1 private
P0 private

- Communication, sharing, synchronization via loads/stores to shared variables
- Facilities for address translation between local/global address spaces
- Requires OS support to maintain this mapping

Why Shared Memory?

Pluses

- For applications looks like multitasking uniprocessor
- For OS only evolutionary extensions required
- Easy to do communication without OS

Minuses

- Proper synchronization is complex
- Communication is implicit so harder to optimize
- **Hardware designers must implement shared mem abstraction**
  - This is hard

Result

- Traditionally bus-based Symmetric Multiprocessors (SMPs), and now CMPs are the most success parallel machines ever
- And the first with multi-billion-dollar markets

Thread-Level Parallelism

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
   spew_cash();
}
```

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically
- Example: database/web server (each query is a thread)
  - **accts** is **shared**, can't register allocate even if it were scalar
  - **id** and **amt** are private variables, register allocated to **r1**, **r2**

Synchronization

- Mutual exclusion : locks, …
- Order : barriers, signal-wait, …

- Implemented using read/write/RMW to shared location
  - Language-level:
    - libraries (e.g., locks in pthread)
    - Programmers can write custom synchronizations
  - Hardware ISA
    - E.g., test-and-set

- OS provides support for managing threads
  - scheduling, fork, join, futex signal/wait

We'll cover synchronization in more detail in a couple of weeks

Cache Coherence

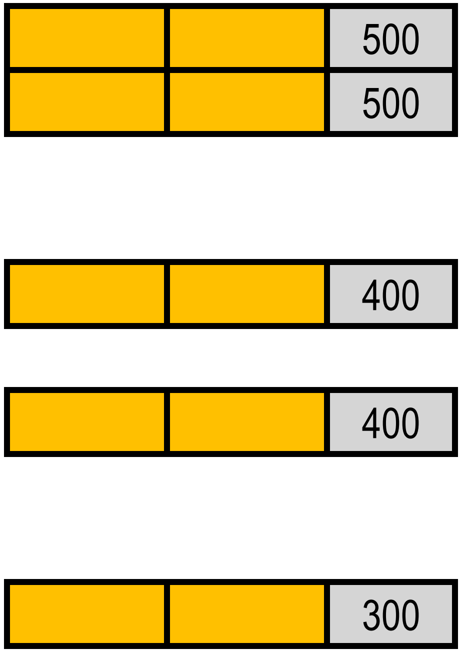
```
Processor 0                   Processor 1
0:  addi r1,accts,r3
1:  ld 0(r3),r4
2:  blt r4,r2,6
3:  sub r4,r2,r4
4:  st r4,0(r3)
5:  call spew_cash            0:  addi r1,accts,r3
                              1:  ld 0(r3),r4
                              2:  blt r4,r2,6
                              3:  sub r4,r2,r4
                              4:  st r4,0(r3)
                              5:  call spew_cash
```

CPU0	CPU1	Mem

- Two $100 withdrawals from account #241 at two ATMs
  - ❒ Each transaction maps to thread on different processor
  - ❒ Track `accts[241].bal` (address is in `r3`)

No-Cache, No-Problem

```
Processor 0               Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash         0: addi r1,accts,r3
                          1: ld 0(r3),r4
                          2: blt r4,r2,6
                          3: sub r4,r2,r4
                          4: st r4,0(r3)
                          5: call spew_cash
```



- Scenario I: processors have no caches
  - No problem

Cache Incoherence

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

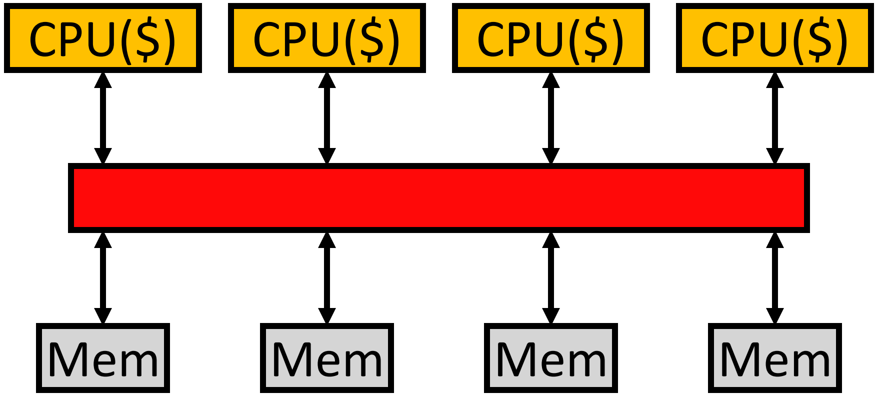
```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

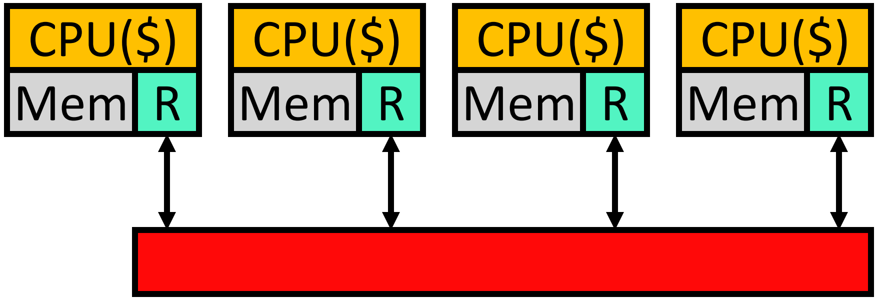
		500
V:500		500
D:400		500
D:400	V:500	500
D:400	D:400	500

- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (out of sync)

Paired vs. Separate Processor/Memory?

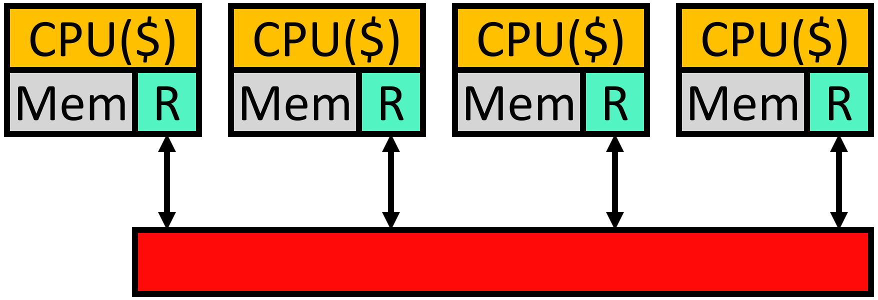
- **Separate processor/memory**
  - ❐ **Uniform memory access (UMA):** equal latency to all memory
  - + Simple software, doesn't matter where you put data
  - – Lower peak performance
  - ❐ Bus-based UMAs common: **symmetric multi-processors (SMP)**
- **Paired processor/memory**
  - ❐ **Non-uniform memory access (NUMA)**: faster to local memory
  - – More complex software: where you put data matters
  - + Higher peak performance: assuming proper data placement

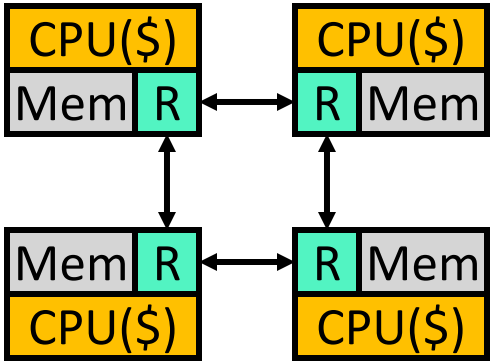




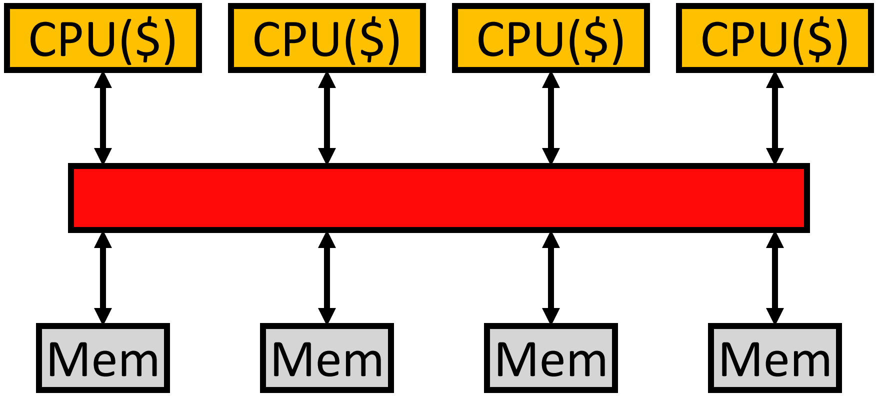
Shared vs. Point-to-Point Networks

- **Shared network**: e.g., bus (left)
  - + Low latency
  - – Low bandwidth: doesn't scale beyond ~16 processors
  - + Shared property simplifies cache coherence protocols (later)
- **Point-to-point network**: e.g., mesh or ring (right)
  - – Longer latency: may need multiple "hops" to communicate
  - + Higher bandwidth: scales to 1000s of processors
  - – Cache coherence protocols are complex



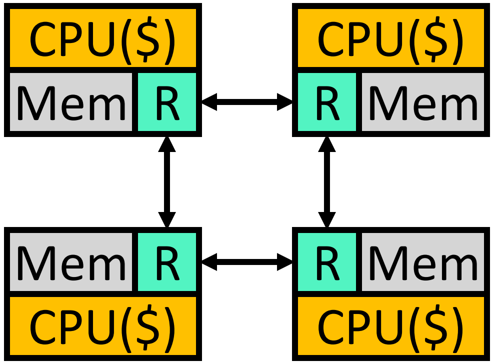


Implementation #1: Snooping Bus MP



- Two basic implementations
- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - Sometimes **multiple processors on single chip (CMP)**
    - **Symmetric multiprocessors (SMPs)**
    - Common, I use one everyday

Implementation #2: Scalable MP



- General point-to-point network-based systems
  - ❐ Typically processor/memory/router blocks (NUMA)
    - ❍ **Glueless MP**: no need for additional “glue” chips
  - ❐ Can be arbitrarily large: 1000’s of processors
    - ❍ **Massively parallel processors (MPPs)**
  - ❐ In reality only government (DoD) has MPPs…
    - ❍ Companies have much smaller systems: 32–64 processors
    - ❍ **Scalable multi-processors**

Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
    - invalidate
    - update
    - supply value
- depends on state of the block and the protocol

Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

- Part I: **bus bandwidth**
  - ❐ Replace non-scalable bandwidth substrate (bus)…
  - ❐ …with scalable bandwidth one (point-to-point network, e.g., mesh)

- Part II: **processor snooping bandwidth**
  - ❐ Interesting: most snoops result in no action
  - ❐ Replace non-scalable broadcast protocol (spam everyone)…
  - ❐ …with scalable **directory protocol** (only spam processors that care)

- We will cover this in Unit 3

Shared Memory Summary

- Shared-memory multiprocessors
  - + “Simple” software: easy data sharing, handles both DLP & TLP
    - …but hard to get fully correct!
  - – Complex hardware: must provide illusion of global address space

- Two basic implementations
  - ❒ **Symmetric (UMA) multi-processors (SMPs)**
    - ❍ Underlying communication network: bus (ordered)
    - + Low-latency, simple protocols
    - – Low-bandwidth, poor scalability
  - ❒ **Scalable (NUMA) multi-processors (MPPs)**
    - ❍ Underlying communication network: point-to-point (often unordered)
    - + Scalable bandwidth
    - – Higher-latency, complex protocols