Overview: Classical Parallel Hardware

Review of Single Processor Design

- so we talk the same language
- many things happen in parallel even on a single processor
- identify potential issues that (explicitly) parallel hardware can overcome
- why should we use 2 CPUs instead of doubling the speed on one!

Multiple Processor Design

- Flynn's taxonomy of parallel computers (SIMD vs MIMD)
- message-passing versus shared-address space programming
- UMA versus NUMA shared-memory computers
- dynamic/static connectivity
- evaluating static networks
- case study: the NCI Gadi supercomputer

The Processor

Performs (among others):

- floating point operations (flops) - add, mult, division (sqrt maybe!)
- integer and logical operations (and, or, etc.)
- instruction processing (fetch, decoding, etc.)
- our primary focus will be in flops (as per required by most scientific applications)
- main performance metric: flops/sec or just FLOPS

The processor clock orchestrates its operation:

- all ops take a fixed number of *clock ticks* to complete (latency)
- clock speed is measured in GHz (10^9 cycles/second) or nsec (10^{-9} seconds)
  - Apple iPhone 6 ARM A8 1.4GHz (0.71ns), NCI Gadi Intel Xeon Cascade Lake 3.2GHz (0.31ns), IBM zEC12 processor 5.5Ghz (0.18ns)
- clock speed limited by: transistor speed, speed of light, energy consumption, etc.
  - (to our knowledge) IBM zEC12 is fastest commodity processor at 5.5GHz
  - light travels about 1cm in 3.2ns, a chip is a few cm!

Processor Performance

flops/sec	Prefix	Occurrence (as of today)
10^3	kilo (k)	very badly written code
10^6	mega (m)	badly written code
10^9	giga (g)	single-core
10^{12}	tera (t)	supercomputer node
10^{15}	peta (p)	all machines in Top500 (Nov 22, measured)
10^{18}	exa (e)	2022!

How peak flops/sec. is computed?

- Desktop 2.5GHz Quad-Core, 4(core)*4(flops)*2.5GHz $\equiv$ 40 gflops/sec.
- Bunyip cluster Pentium III, 96(nodes)*2(sockets)*1(core)*1(flop)*550MHz $\equiv$ 105 gflops/sec,
- NCI Raijin 3592(nodes)*2(sockets)*8(core)*8(flops)*2.6GHz $\equiv$ 1.19 pflops/sec.
- NCI Gadi 3074(nodes)*2(sockets)*24(core)*16(flops)*3.2GHz $\equiv$ 7.55 pflops/sec.

Illustrating pipelining with an example: Adding Float64 Numbers

Consider adding two double precision (8 byte) numbers

0	1 11	12 63
±	Exponent	Significand

Possible steps:

- determine largest exponent
- normalize significand of the smaller exponent to the larger
- add significand
- re normalize the significand and exponent of the result

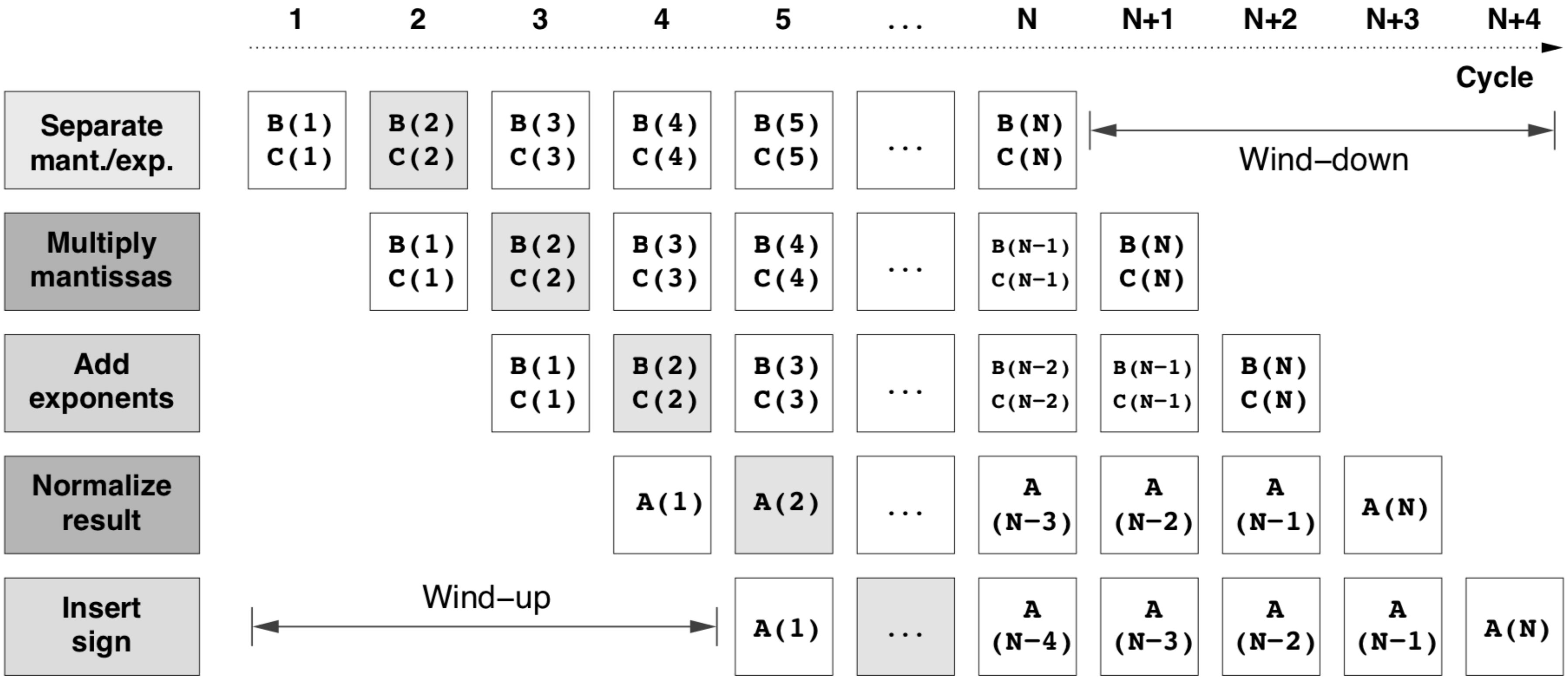
Let us assume each step take 1 clock tick, i.e., a latency of 4 ticks per addition (flop)

Illustrating pipelining with an example: Adding Float64 Numbers

	Step in Pipeline				
Waiting	1	2	3	4	Done
X(6)					
	X(5)→				
		X(4)→			
			X(3)→		
				X(2)→	
					X(1)

- X(1) takes 4 clock ticks to appear (startup latency); X(2) appears 1 tick after X(1)
- asymptotically achieves 1 result per tick
- the operation (X) is said to be pipelined: *steps in the pipeline are running in parallel*
- requires same op consecutively on different (independent) data items
    - good for “vector operations” (note limitations on chaining output data to input)

Another example: Multiplying Float64 Numbers



Cycle	1	2	3	4	5	...	N	N+1	N+2	N+3	N+4
Separate mant./exp.	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	B(N) C(N)	Wind–down			
Multiply mantissas		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	B(N−1) C(N−1)	B(N) C(N)			
Add exponents			B(1) C(1)	B(2) C(2)	B(3) C(3)	...	B(N−2) C(N−2)	B(N−1) C(N−1)	B(N) C(N)		
Normalize result				A(1)	A(2)	...	A(N−3)	A(N−2)	A(N−1)	A(N)	
Insert sign	Wind–up				A(1)	...	A(N−4)	A(N−3)	A(N−2)	A(N−1)	A(N)

Figure 1.5: Timeline for a simplified floating-point multiplication pipeline that executes `A(:)=B(:)*C(:)`. One result is generated on each cycle after a four-cycle wind-up phase.

Instruction Pipelining (Single Instruction Issue)

- break instructions into k stages each that are overlapped in time
- eg. ($k = 5$): stages FI = Fetch Instrn., DI = Decode Instrn., FO = Fetch Operand, EX = Execute Instrn., WB = Write Back

(branch):	FI	DI	FO	EX	WB				
(guess)		FI	DI	FO	EX	WB			
(guess)			FI	DI	FO	EX	WB		
(guess)				FI	DI	FO	EX	WB	
(sure)					FI	DI	FO	EX	WB

- Ideally, one gets k-way asymptotic parallelism (speedup)
- However, hard to maximize utilization in practice:
  - Constrained by dependencies among instructions; CPU must ensure result is the same as if no pipelining!
  - FO & WB stages may involve memory accesses (and may possibly stall the pipeline)
  - conditional branch instructions are problematic: the wrong guess may require flushing succeeding instructions from the pipeline and rolling back
- tendency to increase # of stages (specially acute during 90s-20s)

examples of #stages: UltraSPARC II (9) and III (14), Intel Prescott (31)

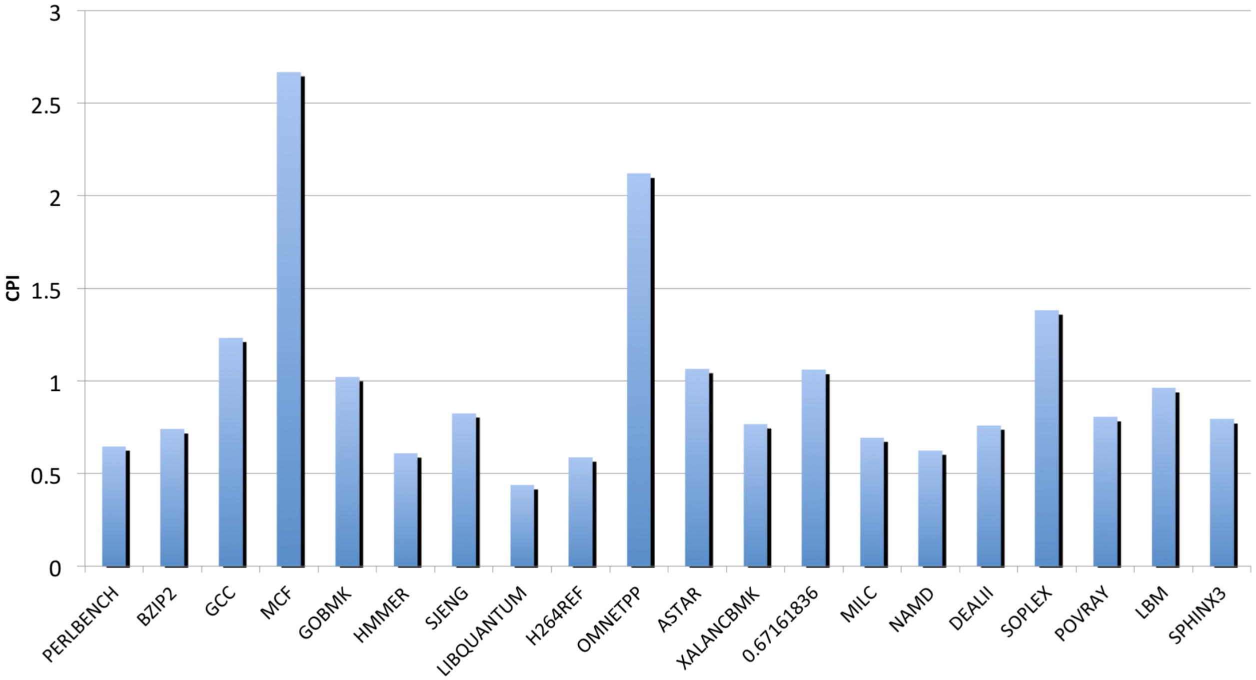
Superscalar execution (Multiple Instruction Issue)

- Simple idea: Increase execution rate by using $w \geq 2$ (i.e., multiple) pipelines
- w (mutually independent) instructions are (tried to be) piped in parallel at each cycle
- Ideally it offers kw-way parallelism (recall k is the number of pipeline stages)
- However, a number of extra challenges arise:
  - Increased complexity: HW has to be able to resolve dependencies at runtime before issuing simultaneously several instructions
  - Some of the functional units might be shared by the pipelines (aka resource dependencies)
  - As a result, instructions to be issued together must have an appropriate 'instruction mix'

    e.g. UltraSPARC ($w = 4$): $\left\{ \begin{array}{l} \leq 2 \text{ different floating point} \\ \leq 1 \text{ load / store ; } \leq 1 \text{ branch} \\ \leq 2 \text{ integer / logical} \end{array} \right.$
- Some remedies: pipeline feedback, branch prediction + speculative execution, out-of-order execution, compilers (e.g., VLIW processors)

Limitations of Instruction-Level Parallelism (ILP)

Actual clock Cycles Per Instruction (CPI) on Intel i7 (Peak is 0.25)



CPU does a lot of (wasted) work that can just not be written back due to branch mispredictions

Limitations of Memory System Performance

Consider the DAXPY computation:

$$y(i) = y(i) + 1.234 * x(i)$$

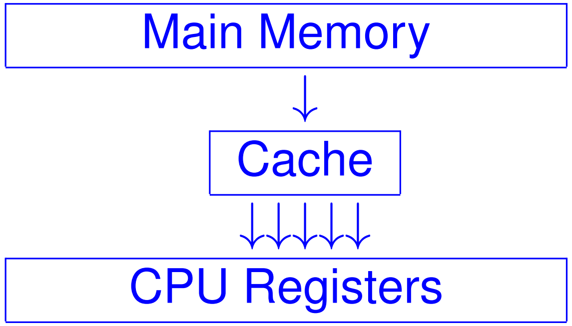
If at its peak the CPU can perform 8 flops/cycle (4 fused mult-add)

- the memory system must load 8 **`doubles`** ($x(i)$ and $y(i)$ – 64 bytes) and store 4 ($y(i)$ – 32 bytes) each clock cycle
  - on a 2 GHz system this implies a memory system able to sustain 128 GB/s load traffic and 64 GB/s store traffic
- despite advances in memory technology (e.g., DDR5 SDRAM), memory is not able to pump data at such high rates

Memory latency and bandwidth are critical performance issues

- caches: reduce latency and provide improved cache to CPU bandwidth
- multiple memory banks: improve bandwidth (by parallel access)

Memory Hierarchy



Main Memory ⟶ large, cheap memory; large latency/small bandwidth

Cache ⟶ small, expensive memory; lower latency/higher bandwidth

- memory is partitioned into blocks (cache lines) and mapped to cache lines using a mapping algorithm (e.g., completely associative, direct, n-way associative)
- cache lines are typically 16-128 bytes wide; entire cache lines fetched from memory, not just one element (why?)
- cache hit (few cycles)/cache miss (large number of cycles)
- try to structure code to use an entire cache line of data before replacement (e.g., blocking strategies in dense matrix-matrix multiplication)

Cache memory is effective because algorithms often use data that:

- was recently accessed from memory (temporal locality)
- was close to other recently accessed data (spatial locality)

Going (Explicitly) Parallel

- performance of a single processor is irremediably limited by clock rate
- clock rate in turn limited by power consumption, transistor switching time, etc.
- ILP allows multiple instructions at once, but it is limited by dependencies
- many problems are inherently distributed/exhibit potential parallelism

It's time to go (explicitly) parallel

Parallel Hardware Overview

- Flynn's Taxonomy of parallel processors (1966,1972)
  - (SISD/SIMD/)SIMD/MIMD
- message-passing versus shared-address space programming
- UMA versus NUMA shared-memory computers
- dynamic/static networks
- evaluating cost and performance of static networks
- case study: NCI's Gadi (2020–)

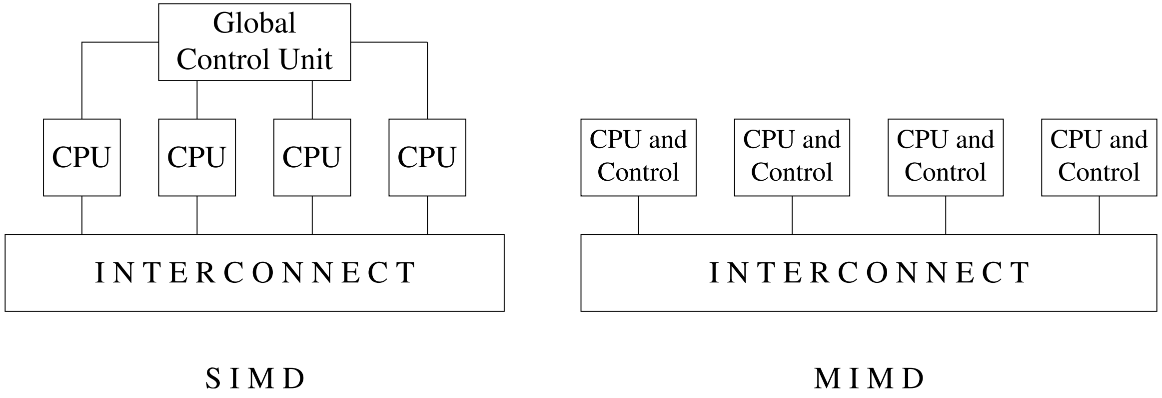
SIMD and MIMD in Flynn's Taxonomy

SIMD: Single Instruction Multiple Data

- also known as data parallel or vector processors (very popular in the 70s and 80s)
- nowadays come mainly in the form of SSE co-processing instructions
- other examples: GPUs; SPEs on Sony's PS3 IBM CellBE (2006)
- perform their best with structured (regular) computations (e.g., image processing)

MIMD: Multiple Instruction Multiple Data

- examples include: (1) quad-core PC; (2) 2x24-core Xeon CPUs on each Gadi node



MIMD

Most successful model for parallel architectures

- more general purpose than SIMD, can be built out of off-the-shelf components
- extra burden to programmer

Some challenges for MIMD machines

- scheduling: efficient allocation of processors to tasks in a dynamic fashion
- synchronization: prevent processors accessing the same data simultaneously
- interconnect design: processor to memory and processor to processor interconnects. Also I/O network - often processors dedicated to I/O devices
- overhead: inevitably there is some overhead associated with coordinating activities between processors, e.g. resolve contention for resources
- partitioning: partitioning a computation/algorithm into concurrent tasks might not be trivial and require algorithm redesign and/or significant programming efforts

Logical classification of parallel computers

Regardless of how they are physically organized under the hood, from a programmer's perspective, parallel computers can be classified into two broad categories:

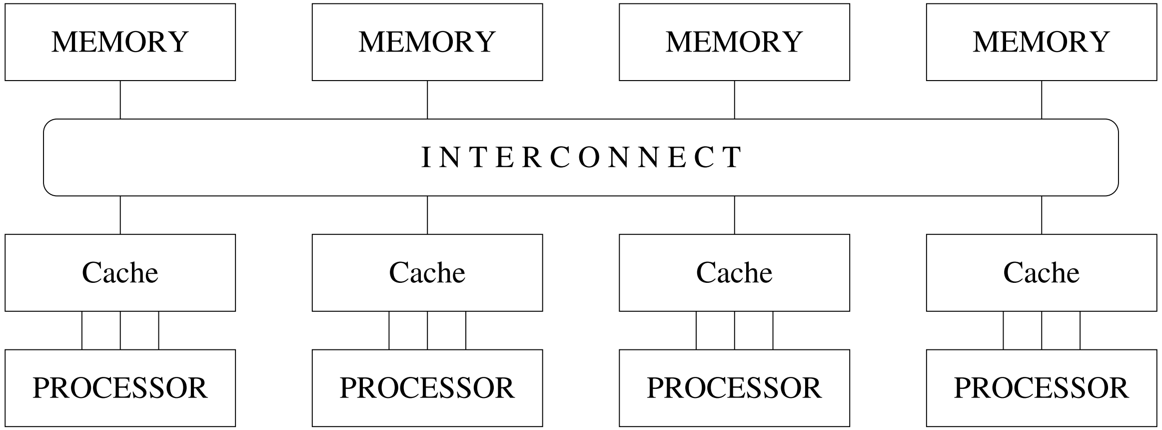
- Message-passing (distributed address space) parallel computers
- Shared address space parallel computers

Address Space Organization: Message Passing

- logically organized as multiple processing nodes, each with its own exclusive/private address space
- interaction among programs running on different nodes accomplished using messages
- messages are used to transfer data, work, and synchronization
- typically implemented in practice by so called distributed memory parallel computers (although not necessarily)
- in these computers, (aggregate) memory bandwidth scales linearly with # of processing nodes
- example: parallelism between "nodes" on the NCI Gadi system

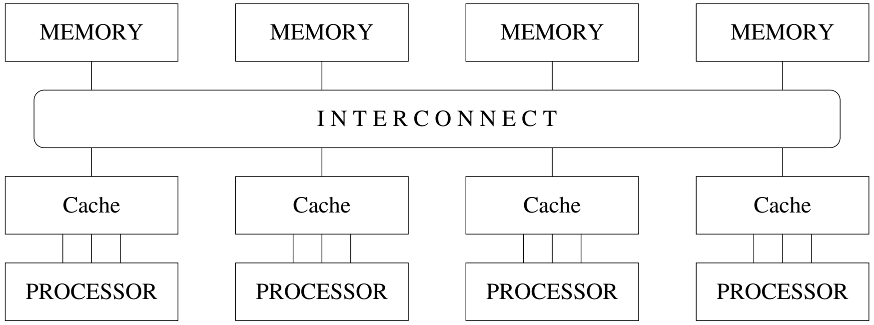
Address Space Organization: Shared Address Space

- there is a common data shared address space
- processes interact by modifying objects stored in this shared address space
- most typically implemented by so-called shared-memory computers
- simplest implementation is a flat or uniform memory access (UMA)
- synchronizing concurrent access to shared data objects and processor-processor communications (to maintain coherence among multiple copies) limits performance
- typically one observes sublinear memory bandwidth with # of processors
- example: QuadCore laptop

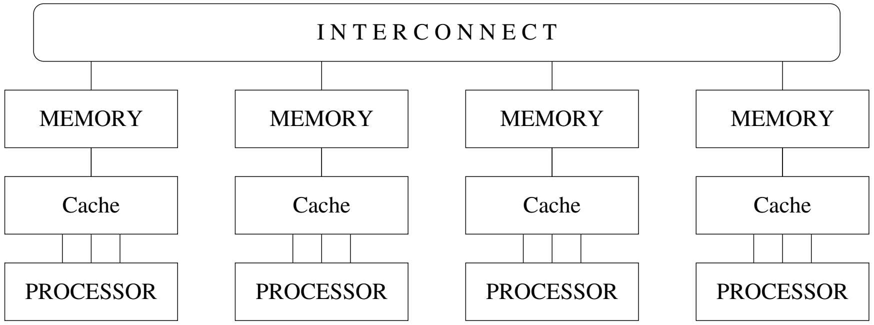


Non-Uniform Memory Access (NUMA)

- all memory is still visible to the programmer (shared address space), but some memory accesses take longer to access than others
- designed to increase aggregated memory bandwidth with # of processors
- parallel programs should be written such that fast memory accesses are maximized (collocate data and computation accordingly)
- example: within each Gadi node, each socket (i.e., 24-core CPU) is connected to its own memory module that is faster to access than the other (remote) one

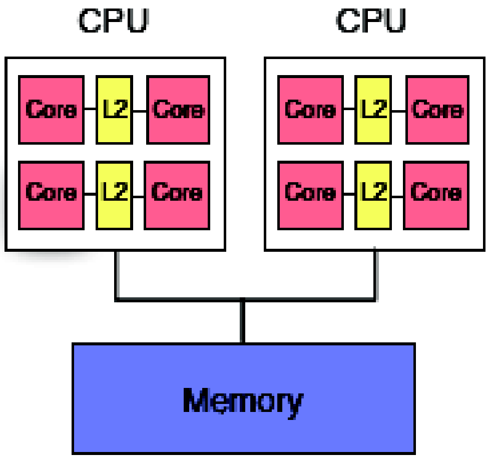


cached UMA

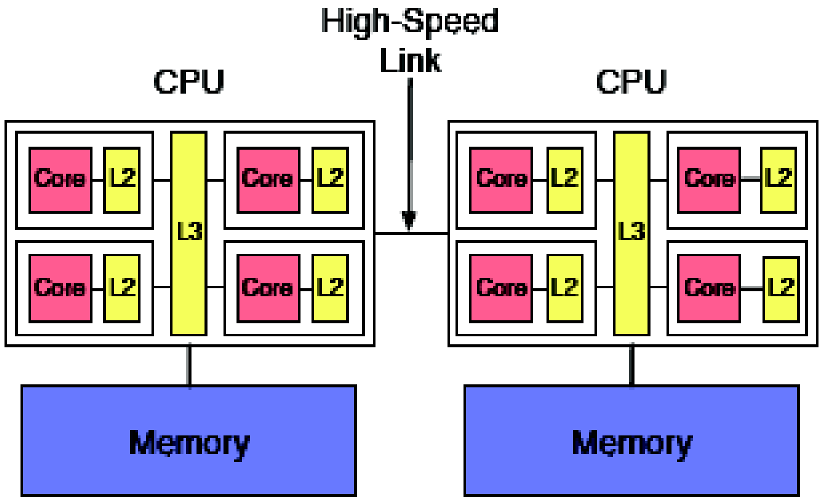


cached NUMA

Another example of shared-memory MIMD computers



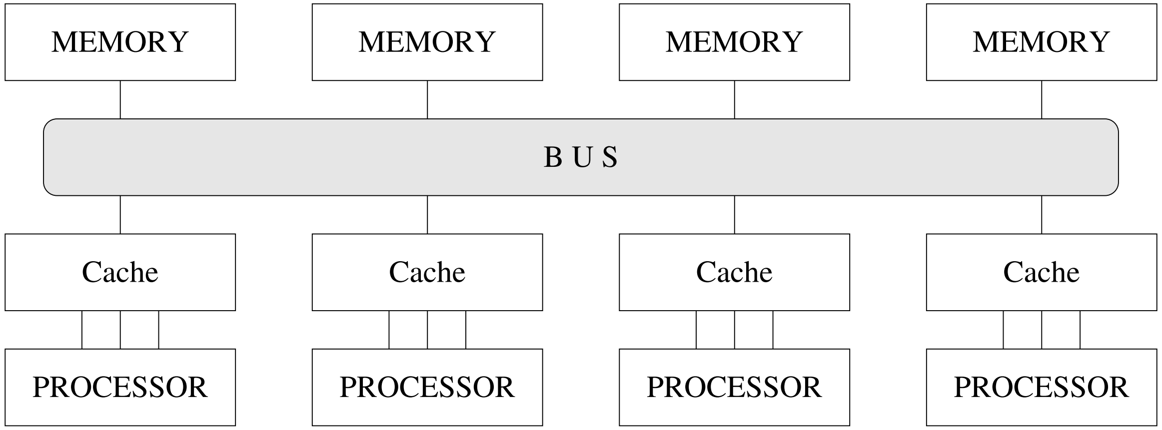
multicore-UMA



multicore-NUMA

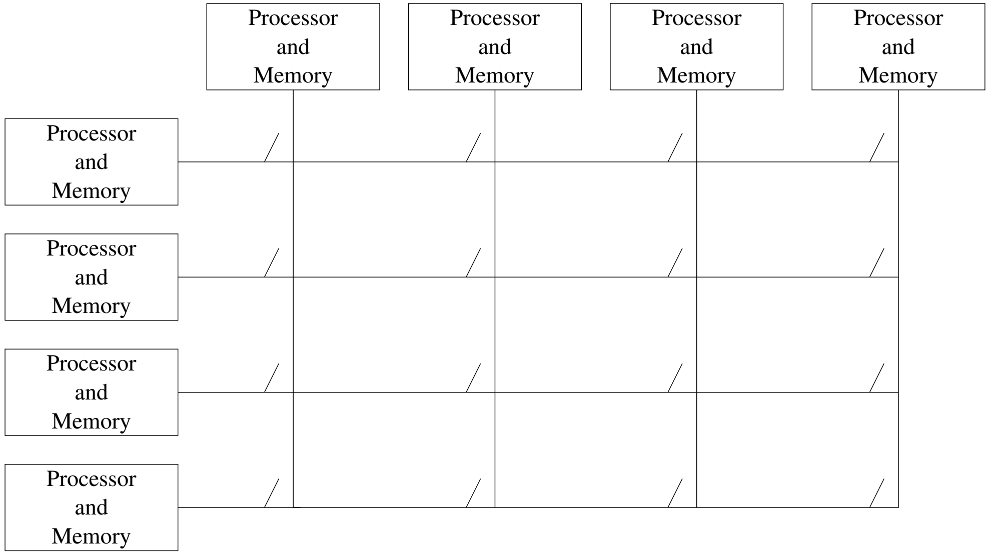
Dynamic Connectivity: Bus

- simplest/cheapest network: shared medium common to all processors
- its a completely-blocking network: a point-to-point comm. among a processor and a memory module, or among processors, prevents any other comm.
- limited bandwidth scalability (multiple accesses to memory are serialized)
- effective cache utilization can alleviate demands on the bus bandwidth

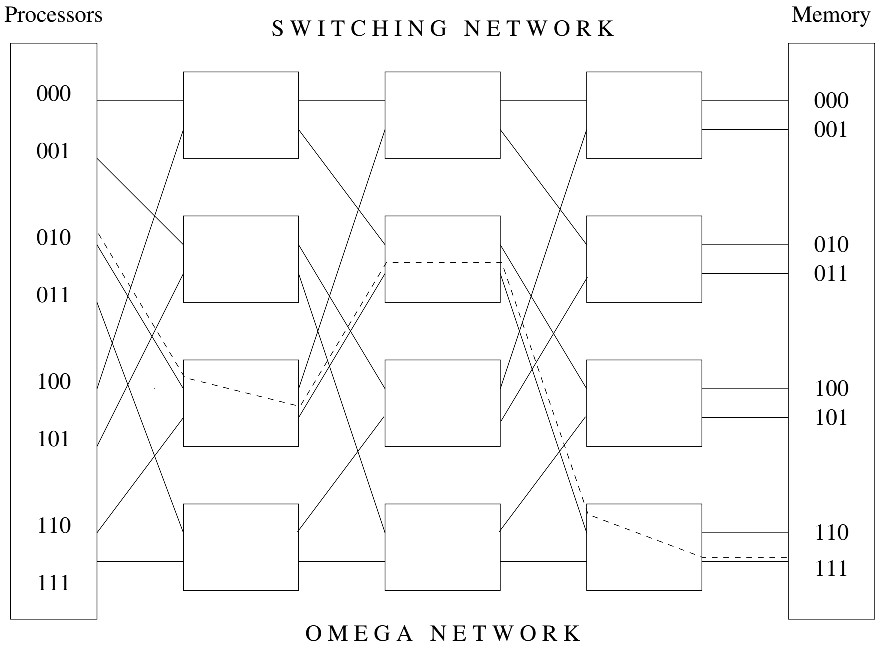


Dynamic Connectivity: Crossbar

- employs a 2D grid of switching nodes (complexity grows as $O(p^2)$)
- its a completely non-blocking network: connection among two processors does not block connection between any other two processors
- not scalable in terms of complexity and cost



Dynamic Connectivity: Multi-staged Networks (e.g. Omega Network)

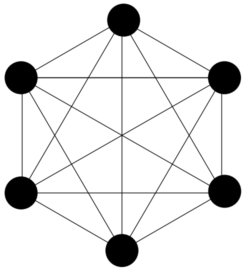


($s = 010$ (src), $t = 111$ (dst), $s \oplus t = 101$)

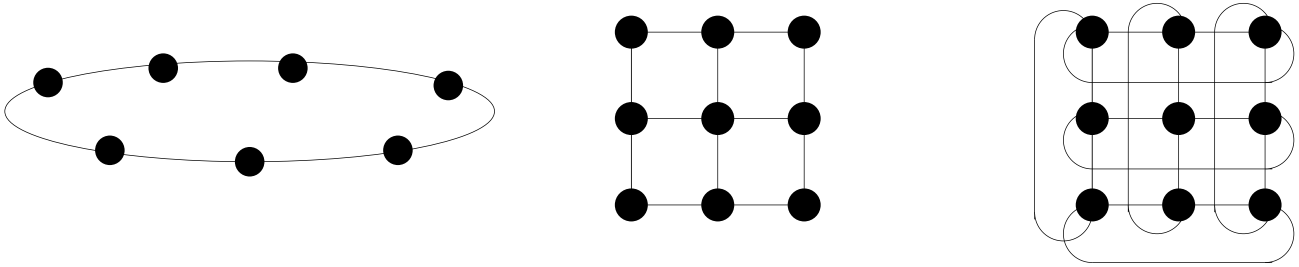
- consists of $\log_2(p)$ stages, $p/2$ switches per stage ($p = 8$ in the figure)
- switches can be configured in two modes: pass-through or crossover
- s and t are binary representations of source and destination
  - processed from most to least significant bit (i.e., left to right)
  - route through if current bits of s and t are the same; otherwise, crossover
- partially blocking network (e.g. consider comms 000-111 and 110-100 at once)

Static Connectivity: Complete, Mesh, Tree

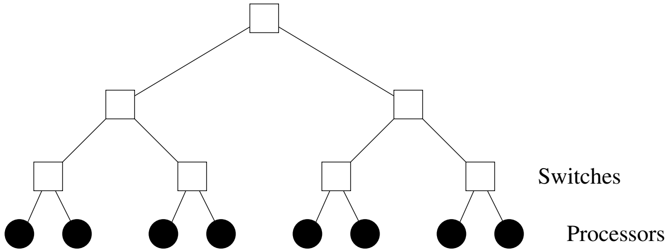
Completely connected (becomes very complex!)



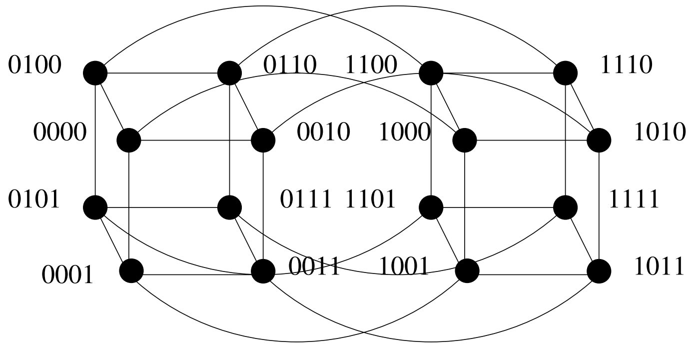
Linear array/ring, mesh/2d torus



Static (all nodes are processors) and dynamic trees (intermediate nodes are switches)



Static Connectivity: Hypercube



$d = 4, \; p = 16$

- two (and exactly two) processing nodes along each dimension, $d = \log_2(p)$ dimensions (thus $p = 2^d$ processing nodes)
- the number of connections per processor grows as $\log_2(p)$
- recursive construction: d-hypercube built by connecting two $d-1$-hypercubes
- two processing nodes directly connected IF ONLY IF their labels differ by one bit
- the number of links in the shortest path between two processors labeled s and t is the number of bits that are on (i.e., =1) in the binary representation of $s \oplus t$ (bitwise XOR) operation (e.g. 3 for $101 \oplus 010$ and 2 for $011 \oplus 101$)
- examples: Intel iPSC Hypercube, NCube, SGI Origin, Cray T3D, TOFU

Evaluating Static Interconnection Networks #1

Diameter

- the maximum distance between any two processors in the network
- directly determines communication time (latency)

Connectivity

- the multiplicity of paths between any two processors
- a high connectivity is desirable as it minimizes contention (also enhances fault-tolerance)
- arc connectivity of the network: the minimum number of arcs that must be removed for the network to break it into two disconnected networks
  - 1 for linear arrays and binary trees
  - 2 for rings and 2D meshes
  - 4 for a 2D torus
  - d for d-dimensional hypercubes

Evaluating Static Interconnection Networks #2

Channel width

- the number of bits that can be communicated simultaneously over a link connecting two processors

Bisection width and bandwidth

- bisection width is the minimum number of communication links that have to be removed to partition the network into two equal halves
- bisection bandwidth is the minimum volume of communication allowed between two halves of the network with equal numbers of processors

Cost

- many criteria can be used; we will use the number of communication links or wires required by the network

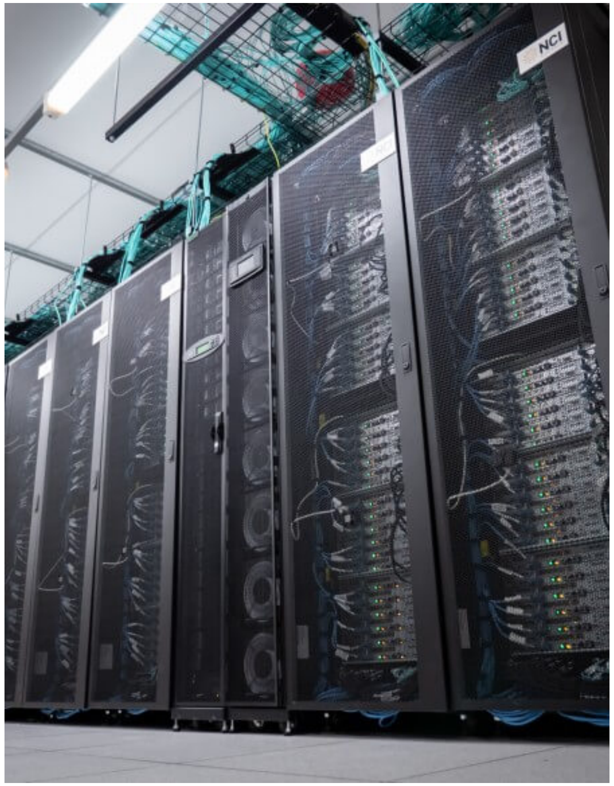
Summary: Static Interconnection Characteristics

Network	Diameter	Bisection width	Arc connectivity	Cost (no. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Binary Tree	$2\log_2((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
Ring	$\lfloor p/2 \rfloor$	2	2	p
2D Mesh	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2D Torus	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log_2 p$	$p/2$	$\log_2 p$	$(p\log_2 p)/2$

Note: the Binary Tree suffers from a bottleneck: all traffic between the left and right sub-trees must pass through the root. The fat tree interconnect alleviates this.

NCI’s Gadi: A Petascale Supercomputer

- 184K cores (dual socket, 24 core Intel Platinum Xeon 8274 (Cascade Lake), 3.2 GHz) in 4243 compute nodes
- 192 GB memory per node (815 TB total)
- Mellanox Infiniband HDR interconnect (100Gbs, ≈ 60 km cables)
  - interconnects: mesh (cores), full (sockets), Dragonfly+ (nodes)
  - ≈ 22 PB Lustre parallel filesystem
  - power: 1.5 MW max. load
  - cooling systems: 100 tonnes of water
  - 24th fastest in the world in debut (June 2020) – 9.3 PFLOPS
    - (probably) fastest file-system in the s. hemisphere
    - custom Linux kernel (CentOS 8)
    - highly customised PBS Pro scheduler



Further Reading: Parallel Hardware

- The Free Lunch Is Over!
- Ch 1, 2.1-2.4 of Introduction to Parallel Computing
- Ch 1, 2 of Principles of Parallel Programming