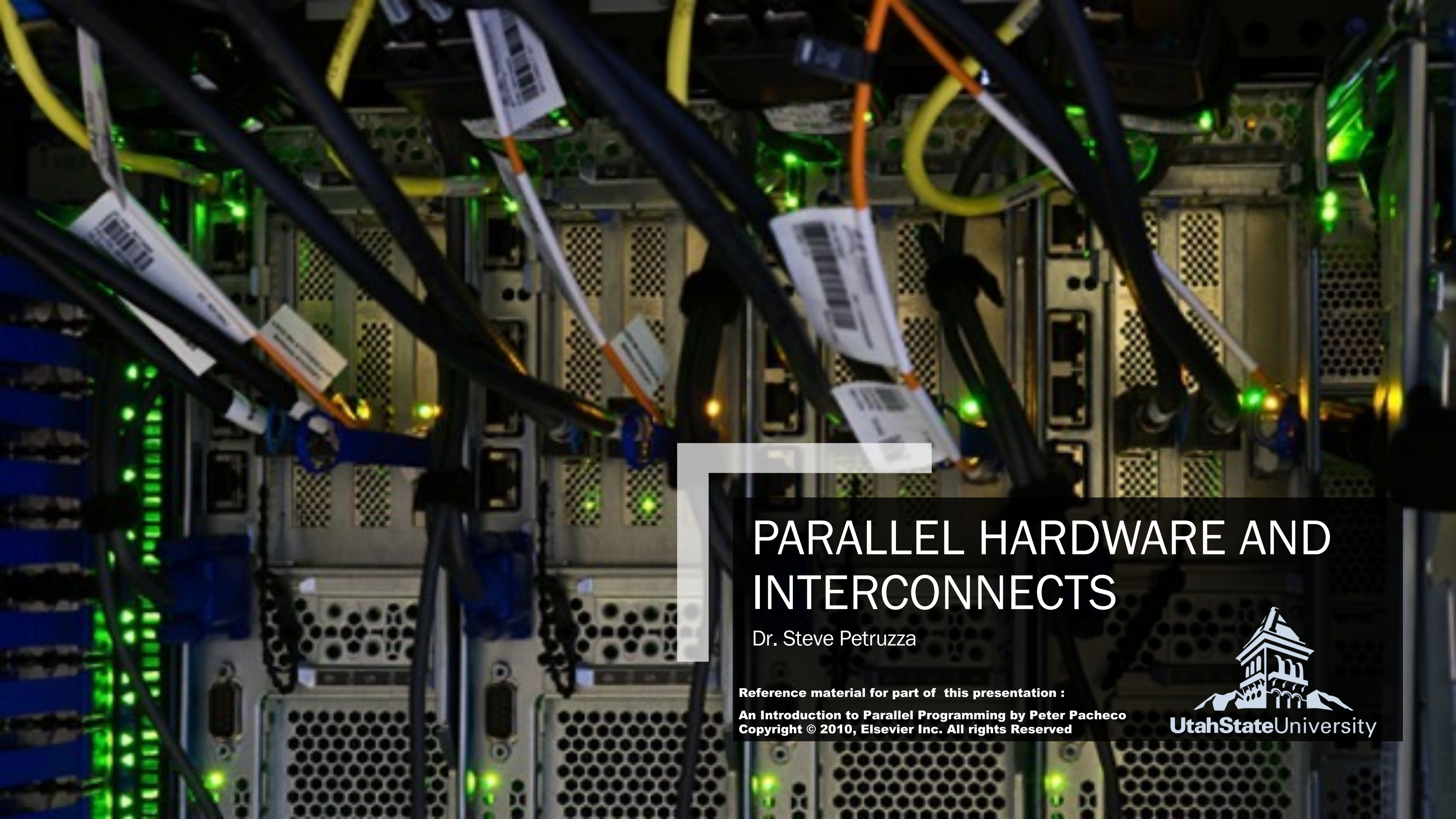


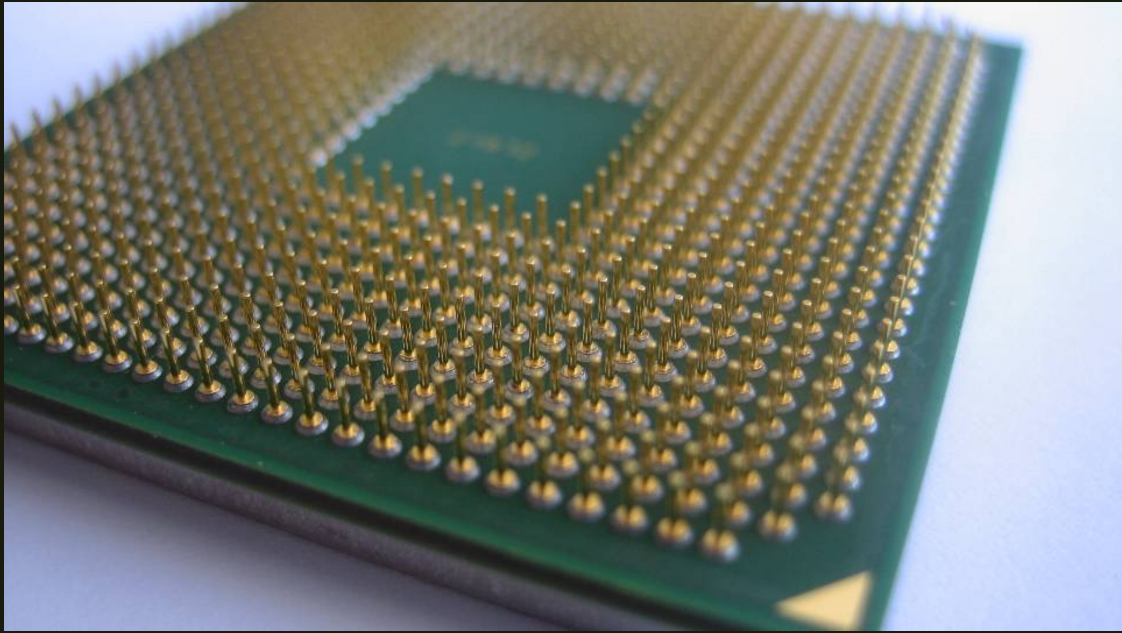
PARALLEL HARDWARE AND INTERCONNECTS

Dr. Steve Petruzza

Reference material for part of this presentation :

An Introduction to Parallel Programming by Peter Pacheco

UtahStateUniversity



How processors execute instructions in parallel

Instruction Level Parallelism

- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

- **Pipelining** - functional units are arranged in stages.

- **Multiple issue** - multiple instructions can be simultaneously initiated.

Pipelining

Similar to a factory assembly line

Pipelining example (1)

Add the floating point numbers
9.87×10^4 and 6.54×10^3

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Pipelining example (2)

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

- Assume each operation takes one nanosecond (10^{-9} seconds).
- This for loop takes about 7000 nanoseconds.

Pipelining (3)

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next.

Pipelining (4)

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Pipelined Addition.

Numbers in the table are subscripts of operands/results.

Pipelining (5)

- One floating point addition still takes 7 nanoseconds.
- But 1000 floating point additions now takes 1006 nanoseconds!

Multiple Issue (1)

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

$$\text{for } (i = 0;\ i < 1000;\ i{+}{+})$$

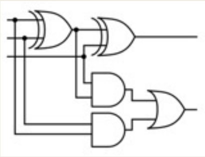
$$z[i] = x[i] + y[i];$$

z[3]

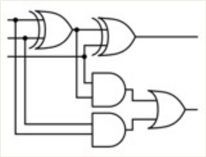
z[4]

z[1]

z[2]



adder #1



adder #2

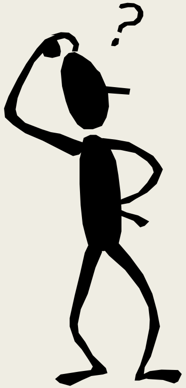
Multiple Issue (2)

- **static** multiple issue - functional units are scheduled at compile time.
- **dynamic** multiple issue – functional units are scheduled at run-time.

superscalar

Speculation (1)

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.



- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess
- How would you make a guess?

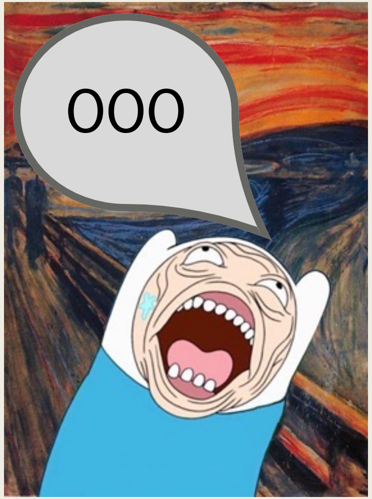
Speculation (2)

```
z = x + y ;
if ( z > 0)
    w = x ;
else
    w = y ;
```

If the system speculates incorrectly,
it must go back and recalculate w = y.

Improving instructions scheduling

- Pipelining still leave some “bubbles” of compute time that can could be still useful
- What can we do to fill up those bubbles?



Out Of Order execution

Group independent set of instructions in the program and execute them out of order (**in hardware**)

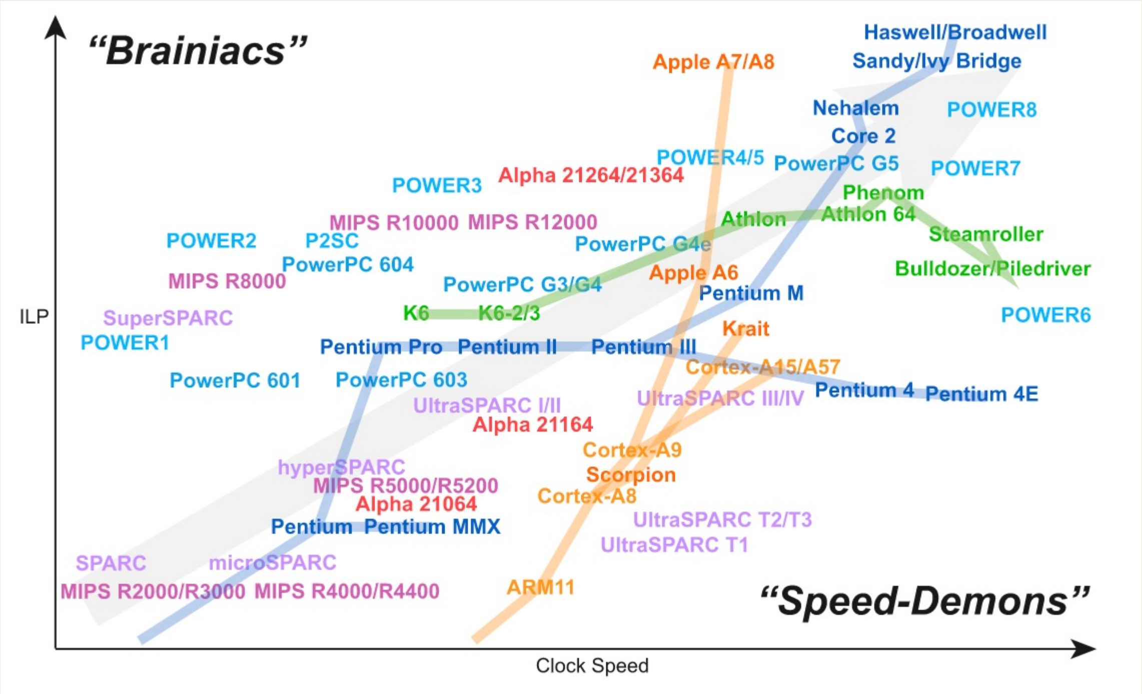
OOO adoption in modern time

- Most of the early superscalars were in-order designs (SuperSPARC, hyperSPARC, UltraSPARC, Alpha 21064 & 21164, the original Pentium).
- Examples of early OOO designs included the MIPS R10000, Alpha 21264 and to some extent the entire POWER/PowerPC line (with their reservation stations).
- Today, **almost all high-performance processors are out-of-order designs**, with the notable exceptions of UltraSPARC III/IV, POWER6 and Denver.
- Most **low-power**, low-performance processors, such as Cortex-A7/A53 and Atom, are **in-order designs** because OOO logic consumes a lot of power for a relatively small performance gain.

The brainiac vs speed-demon debate

- Can compilers do this costly job instead (of leaving it to the hardware)?
- *Brainiac* designs are at the smart-machine end of the spectrum, with lots **of OOO hardware**
- *Speed-demon* designs are simpler and smaller, relying on a **smart compiler**
- **OOO hardware** should make it possible for **more instruction-level parallelism** to be extracted because things will be known at **runtime** that cannot be predicted in advance
- On the other hand, a simpler **in-order design** will be smaller and use **less power**, which means you can place **more small in-order cores** onto the same chip as fewer, larger out-of-order cores
- Which would you rather have: 4 powerful brainiac cores, or 8 simpler in-order cores?

Not a winner yet…



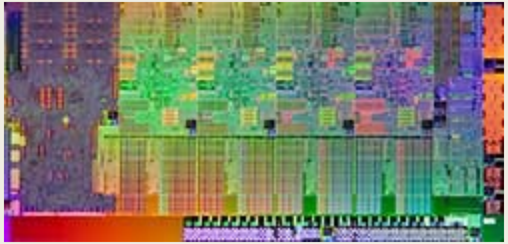
http://www.lighterra.com/papers/modernmicroprocessors/

SMT/Hyper-Threading

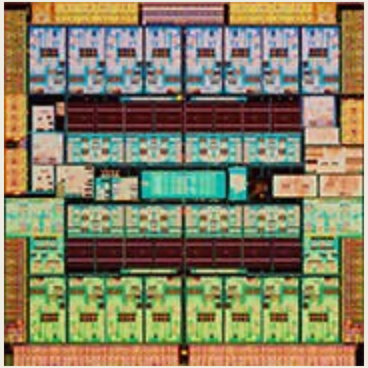
- Superscalar execution is weakened by the fact that many programs simply don't have a lot of fine-grain parallelism
- SMT (Simultaneous Multi-Threading) exploits parallelism in multiple threads within the same program
- An SMT processor will appear a multi-processor system
- SMT requires to duplicate hardware that stores the execution state of each thread (e.g., program counter, some register), but things like cache and functional units are shared between threads
- The Pentium 4 was the first processor to use SMT, which Intel calls "hyper-threading". Its design allowed for 2 simultaneous threads

More cores

- Why build multiple cores if you can just do SMT?
- Multiple issue processors require a much wider chip area for wiring and placing all the components
- Several simpler single-issue cores can fit the size of a typical SMT core (e.g., 12 : 1)



Intel Core i2 Sandy Bridge
4 large 6-issue OOO (brainiac) cores
Each running 2 threads (“8” threads total)

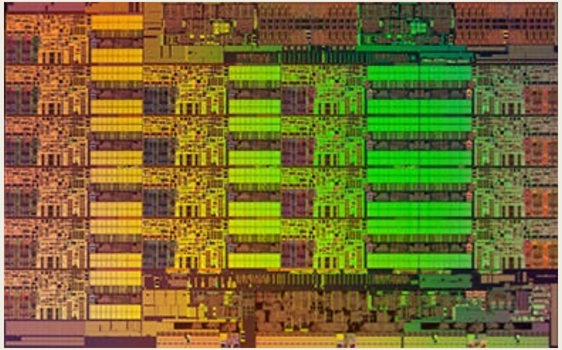


Sun/Oracle UltraSPARC T3 “Niagara 3”
16 small 2-issue in-order cores each running 8 threads
a total of 128 (slower) threads

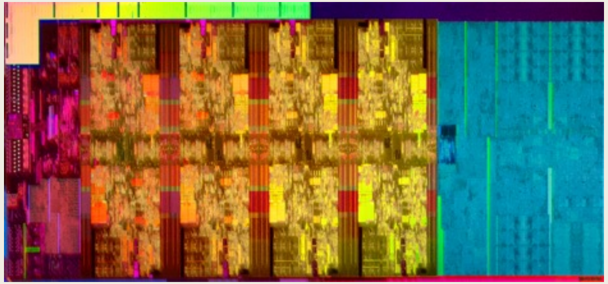
1 billion transistors

Which multi-core approach is better?

- It depends on the application
- Many applications just don't have many threads, better fewer but faster cores
- Some other applications (database systems, 3D rendering) instead can exploit more threads available



Intel Core i4 Xeon Haswell
18 cores with 8-issue design with 2-thread SMT
5.7 billion transistors



Intel Core i9 Coffee Lake
8 cores with 2-thread SMT

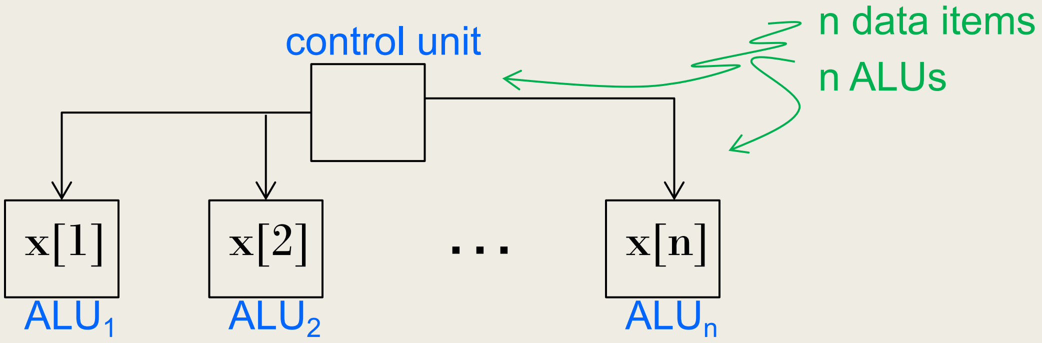
maybe 7 billion transistors

Flynn’s Taxonomy

		Instruction stream	
		Single	Multiple
Data stream	Single	SISD	MISD
	Multiple	SIMD	MIMD

SIMD

- Parallelism achieved by dividing data among the processors.
- Applies the same instruction to multiple data items.
- Called data parallelism.



$$\text{for } (i = 0;\ i < n;\ i{+}{+})$$
$$x[i] \mathrel{+}= y[i];$$

SIMD

- What if we don’t have as many ALUs as data items?
- Divide the work and process iteratively.
- Ex. m = 4 ALUs   and   n = 15 data items.

Round3	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD drawbacks

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- The ALUs have no instruction storage.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.

Vector processors

- Operate on arrays or vectors of data while conventional CPU’s operate on individual data elements or scalars.

- Vector registers.
  - *Capable of storing a vector of operands and operating simultaneously on their contents.*

- Vectorized and pipelined functional units.
  - *The same operation is applied to each element in the vector (or pairs of elements).*

- Vector instructions.
  - *Operate on vectors rather than scalars.*

Vector processors (2)

- Interleaved memory.
  - *Multiple “banks” of memory, which can be accessed more or less independently.*
  - *Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.*
- Strided memory access and hardware scatter/gather.
  - *The program accesses elements of a vector located at fixed intervals.*

Vector processors - Pros

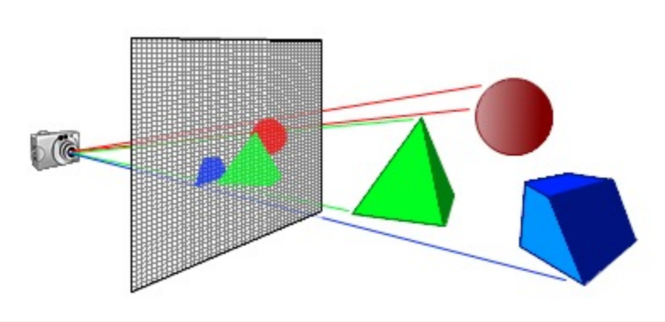
- Fast.
- Easy to use.
- Vectorizing compilers are good at identifying code to exploit.
- Compilers also can provide information about code that cannot be vectorized.
  - *Helps the programmer re-evaluate code.*
- High memory bandwidth.
- Uses every item in a cache line.

Vector processors - Cons

- They don’t handle irregular data structures as well as other parallel architectures.
- A very finite limit to their ability to handle ever larger problems. (scalability)

Graphics Processing Units (GPU)

- Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.
- A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.
- Several stages of this pipeline (called shader functions) are programmable.
  - *Typically just a few lines of C code.*



GPUs

- Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.

- GPU’s can often optimize performance by using SIMD parallelism.
- The current generation of GPU’s use SIMD parallelism.
  - *Although they are not pure SIMD systems.*

GPU vs vector processors, how differently would they execute this code?

```
sum = 0.0;
for (i = 0; i < n; i++) {
  y[i] += a*x[i];
  sum += z[i]*z[i];
}
```

MIMD

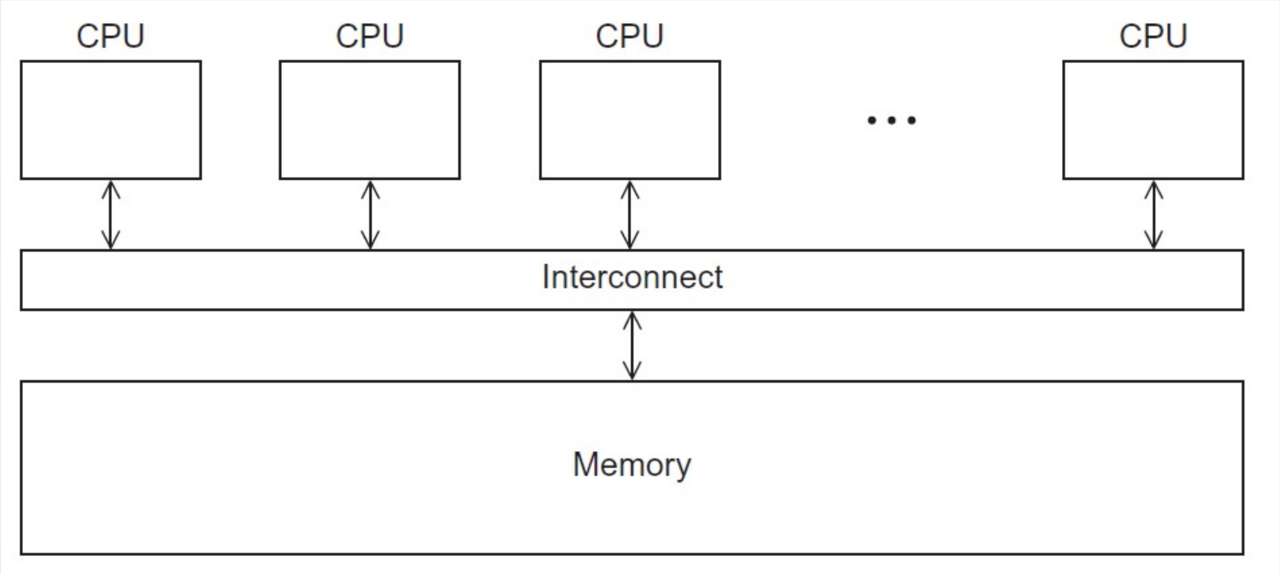
- Supports multiple simultaneous instruction streams operating on multiple data streams.
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

Shared Memory System

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.

- Most widely available shared memory systems use one or more multicore processors.
  - *(multiple CPU's or cores on a single chip)*

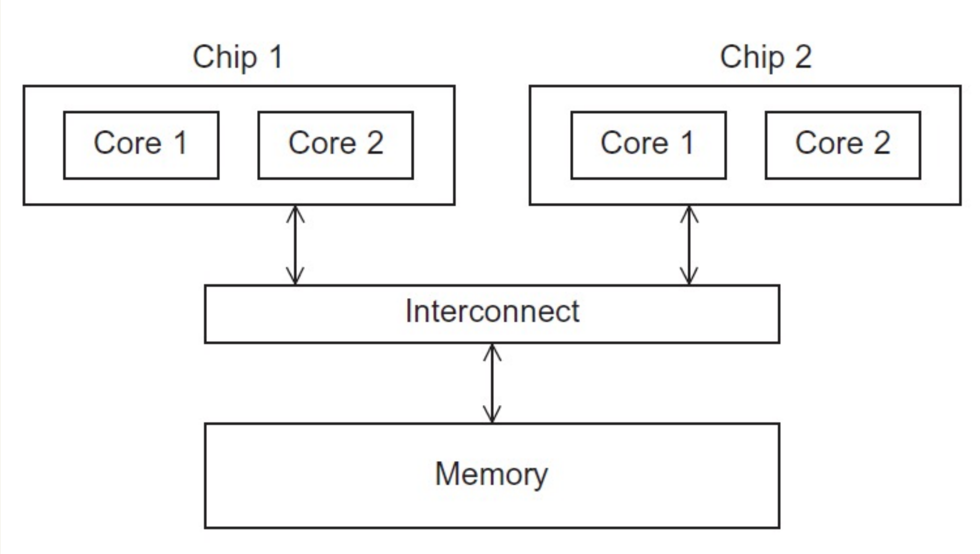
Shared Memory System



UMA and NUMA multicore system

UMA (Uniform Memory Access):
Time to access all
the memory locations
will be the same for
all the cores.

NUMA (Non Uniform Memory Access):
A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

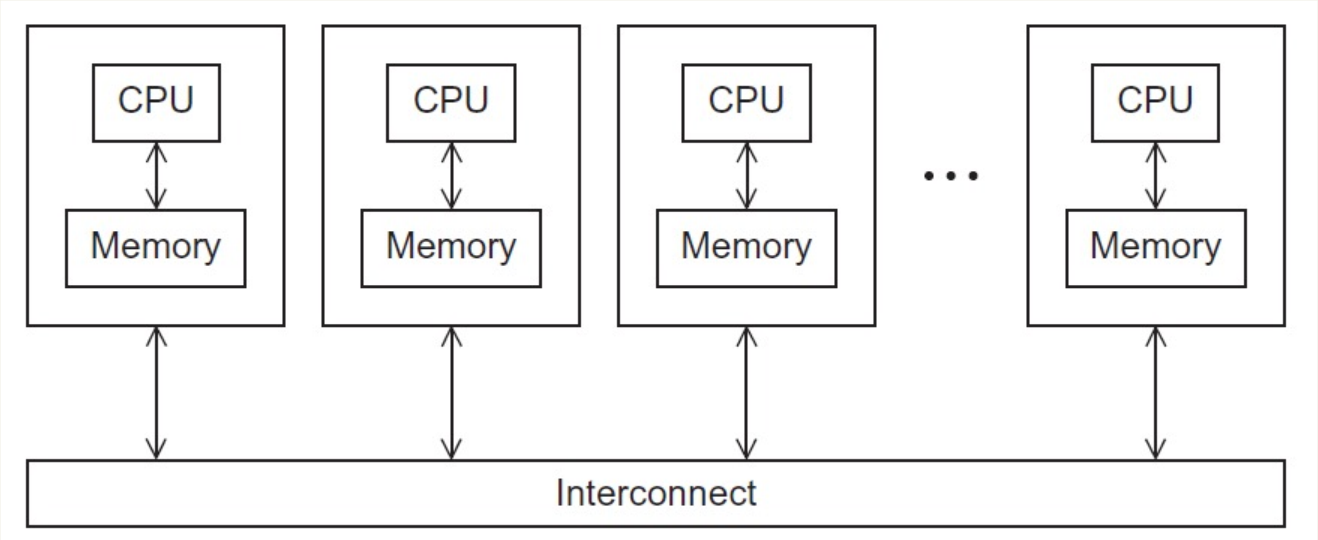


Distributed Memory System

- Clusters (most popular)
  - *A collection of commodity systems.*
  - *Connected by a commodity interconnection network.*

- Nodes of a cluster are individual computations units joined by a communication network.

Distributed Memory System



Cache coherence

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

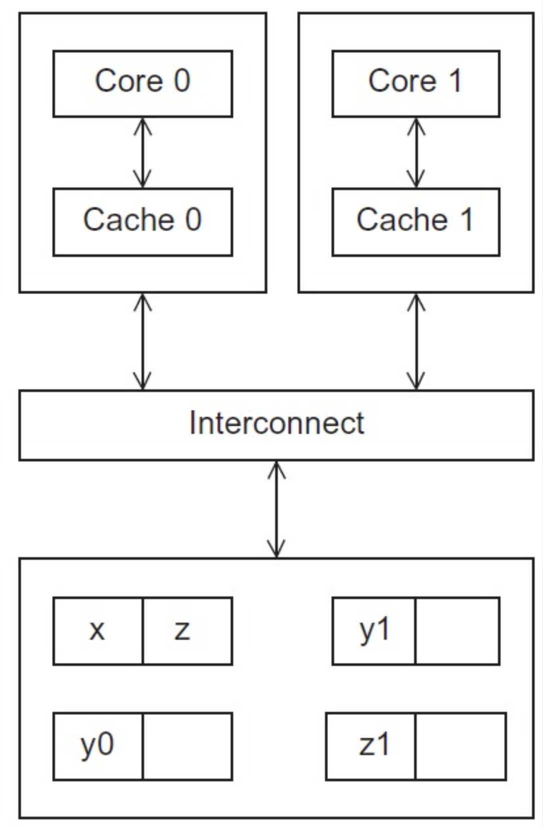
y0 eventually ends up = 2

y1 eventually ends up = 6

z1 = ???

Cache coherence

- Programmers have no control over caches and when they get updated.



A shared memory system with two cores and two caches

Snooping Cache Coherence

- The cores share a bus .
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.

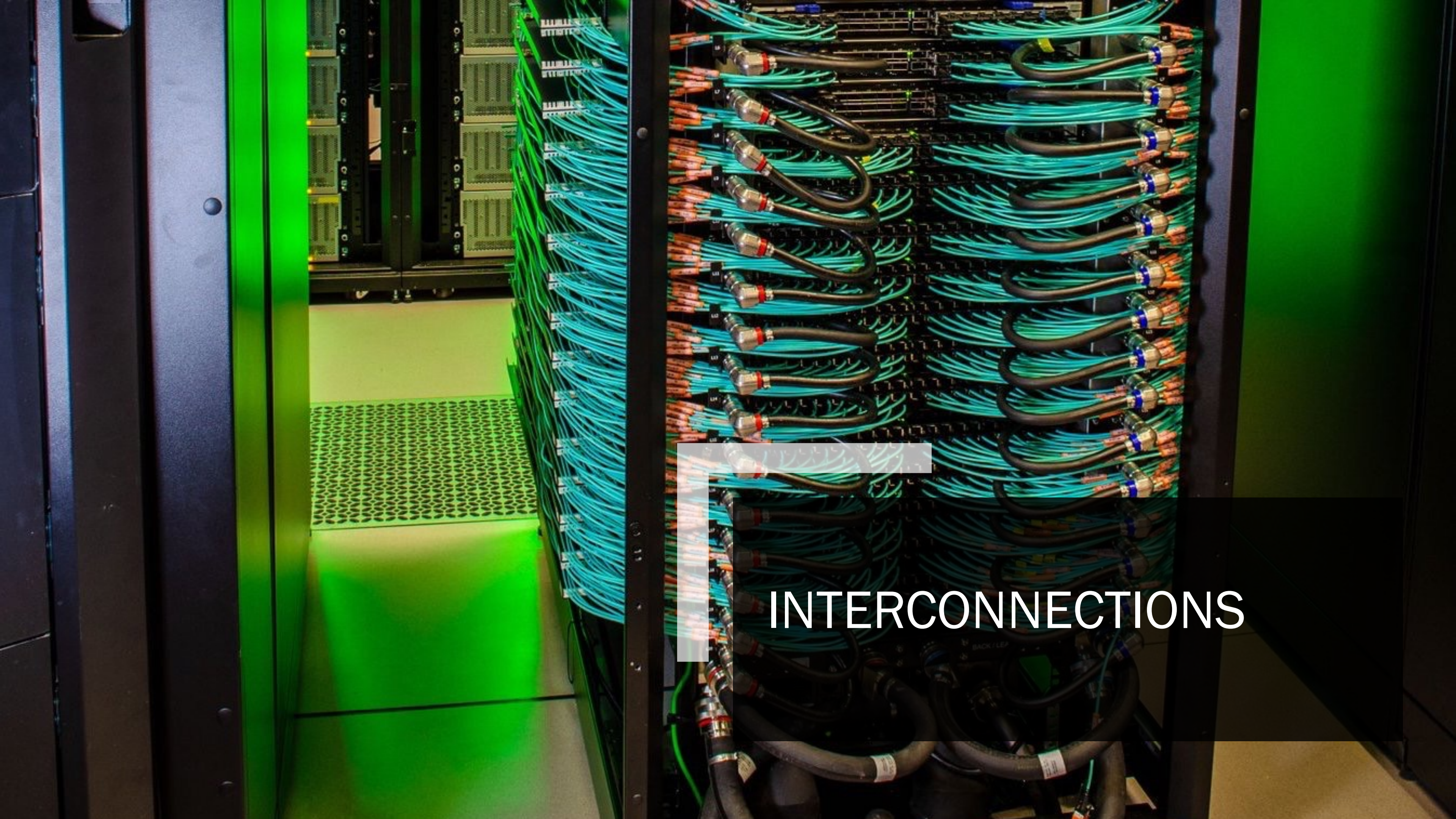
- In large networks snooping is not scalable

Directory based Cache Coherence

- Uses a data structure called a directory that stores the status of each cache line.

- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

INTERCONNECTIONS



Interconnection networks

- Affects performance of both distributed and shared memory systems.

- Two categories:
  - *Shared memory interconnects*
  - *Distributed memory interconnects*

Shared memory interconnects

- Bus interconnect
  - *A collection of parallel communication wires together with some hardware that controls access to the bus.*
  - *Communication wires are shared by the devices that are connected to it.*
  - *As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.*

- Switched interconnect
  - *Uses switches to control the routing of data among the connected devices.*
  - *Crossbar* –
    - Allows simultaneous communication among different devices.
    - Faster than buses.
    - But the cost of the switches and links is relatively high.

Figure 2.7

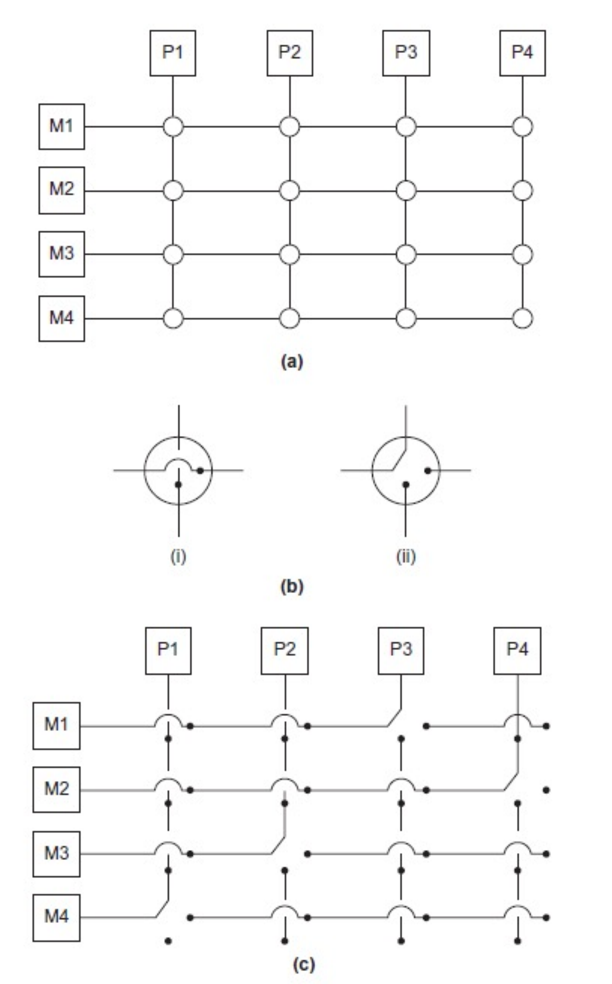
(a)

A crossbar switch connecting 4 processors (P_i) and 4 memory modules (M_j)

(b)

Configuration of internal switches in a crossbar

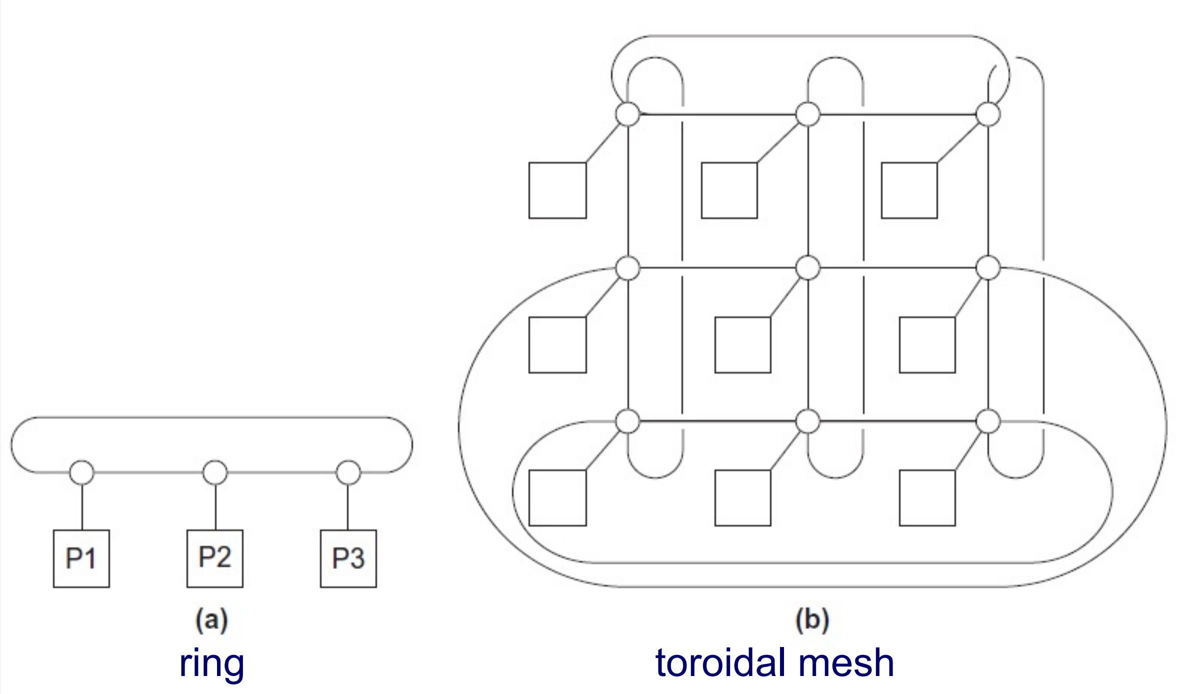
(c) Simultaneous memory accesses by the processors



Distributed memory interconnects

- Two groups
  - *Direct interconnect*
    - Each switch is directly connected to a processor memory pair, and the switches are connected to each other.
  - *Indirect interconnect*
    - Switches may not be directly connected to a processor.

Direct interconnect

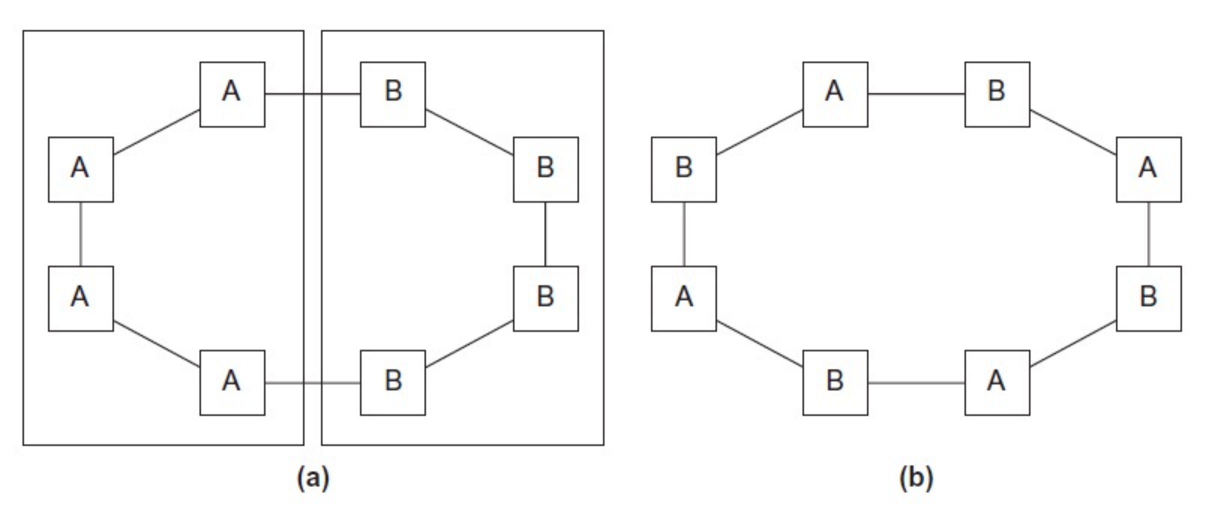


Bisection width

- A measure of "number of **simultaneous communications**" or "**connectivity**".

- How many simultaneous communications can take place "across the divide" between the halves?
- **Bisection Width**: Minimum number of edges that must be removed in order to divide the network into two halves of equal size, or size differing by at most one node.

Two bisections of a ring



Bisection 2 simultaneous communication between halves

4 simultaneous connections

Definitions

- Bandwidth
  - *The rate at which a link can transmit data.*
  - *Usually given in megabits or megabytes per second.*

- Bisection bandwidth
  - *A measure of network quality.*
  - *Instead of counting the number of links joining the halves, it sums the bandwidth of the links.*

More definitions

- Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.
- Latency
  - *The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.*
- Bandwidth
  - *The rate at which the destination receives data after it has started to receive the first byte.*

Message transmission time = $l + n / b$

latency (seconds)

length of message (bytes)

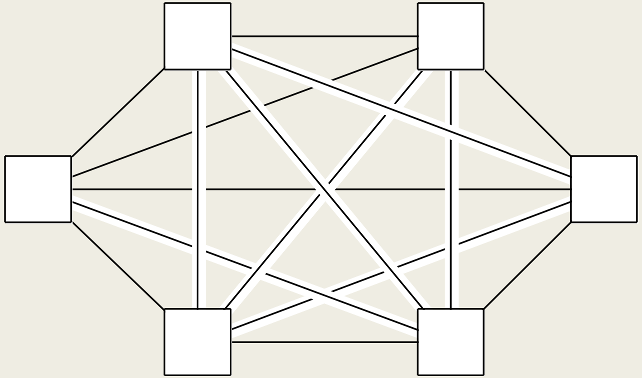
bandwidth (bytes per second)

Fully connected network

- Each switch is directly connected to every other switch.

impractical

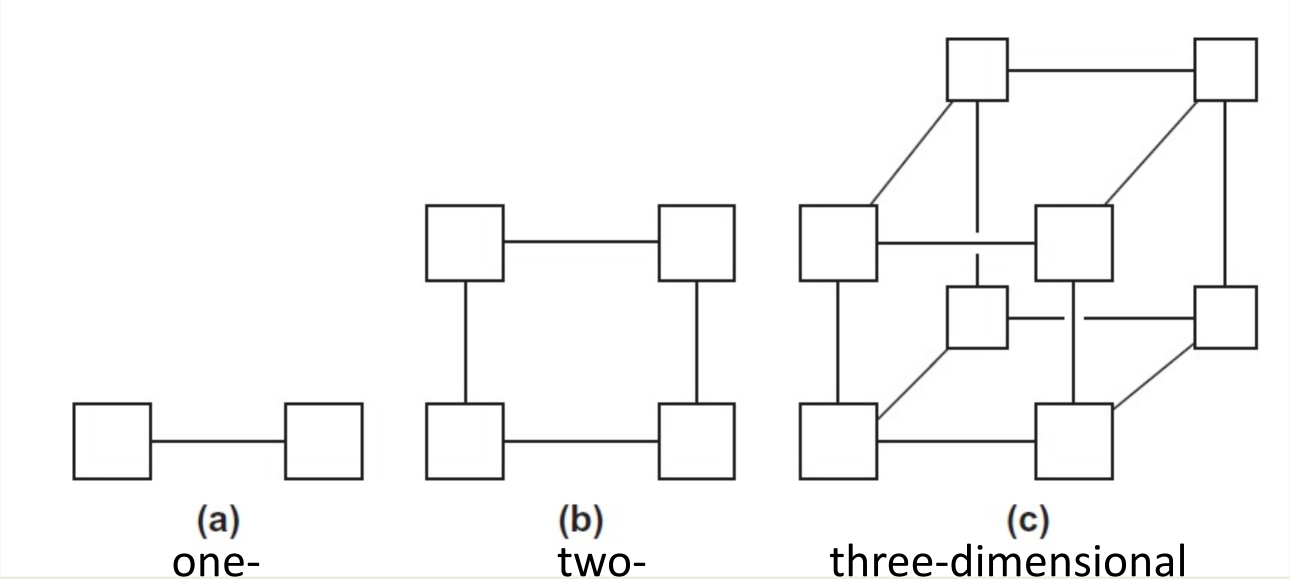
bisection width = $p^2/4$



Hypercube

- Highly connected direct interconnect.
- Built inductively:
  - *A one-dimensional hypercube is a fully-connected system with two processors.*
  - *A two-dimensional hypercube is built from two one-dimensional hypercubes by joining “corresponding” switches.*
  - *Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.*

Hypercubes

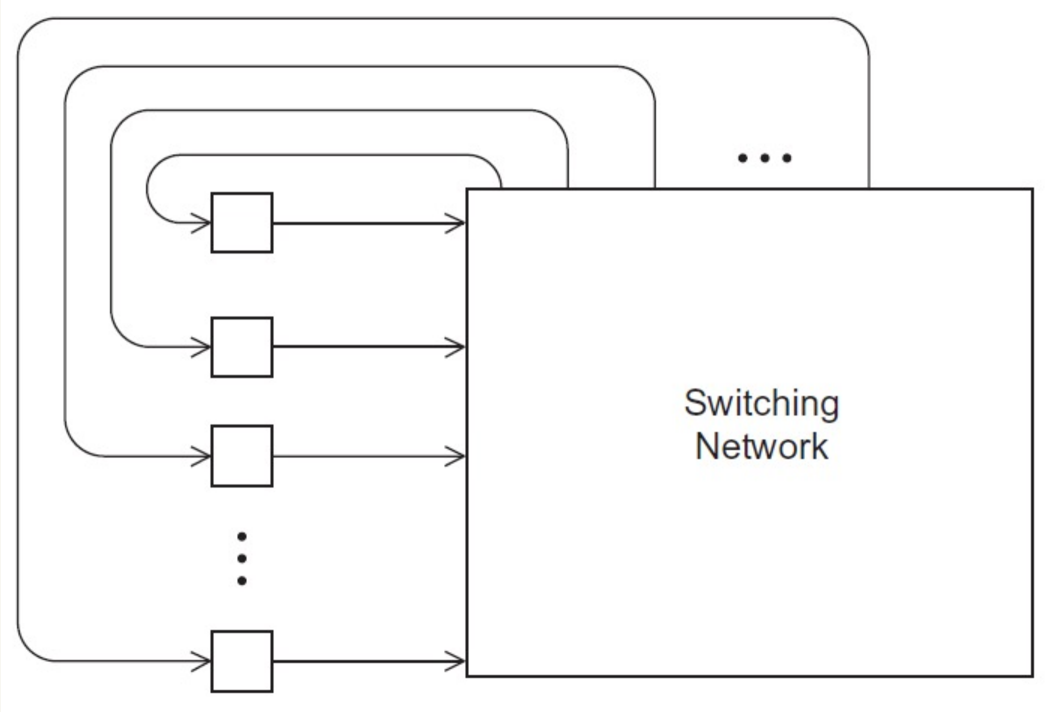


Indirect interconnects

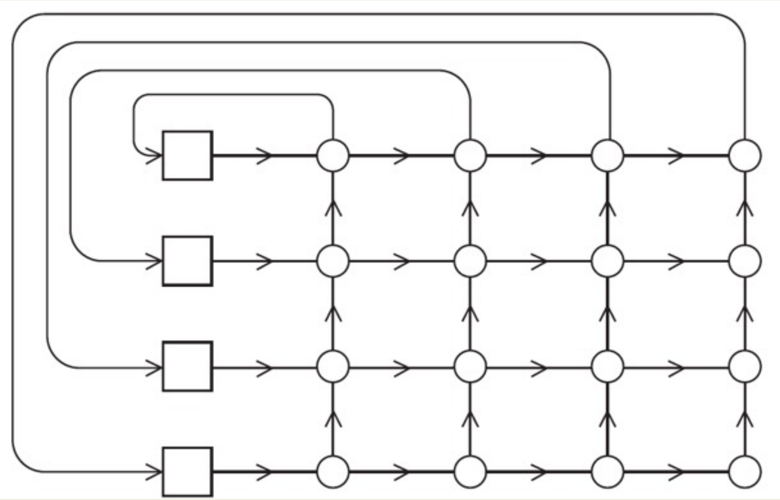
- Simple examples of indirect networks:
  - *Crossbar*
  - *Omega network*

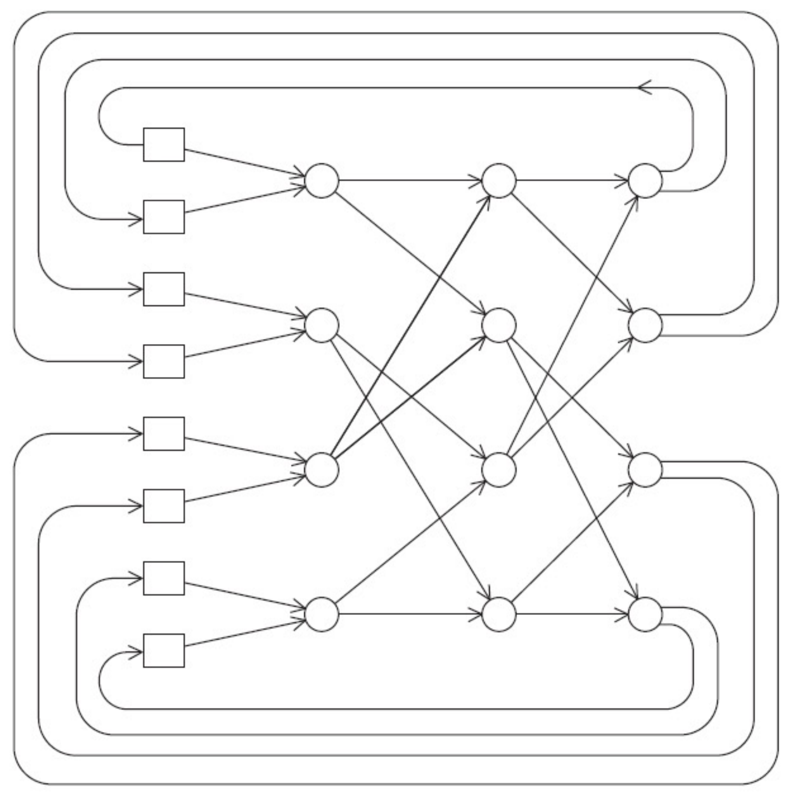
- Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.

A generic indirect network

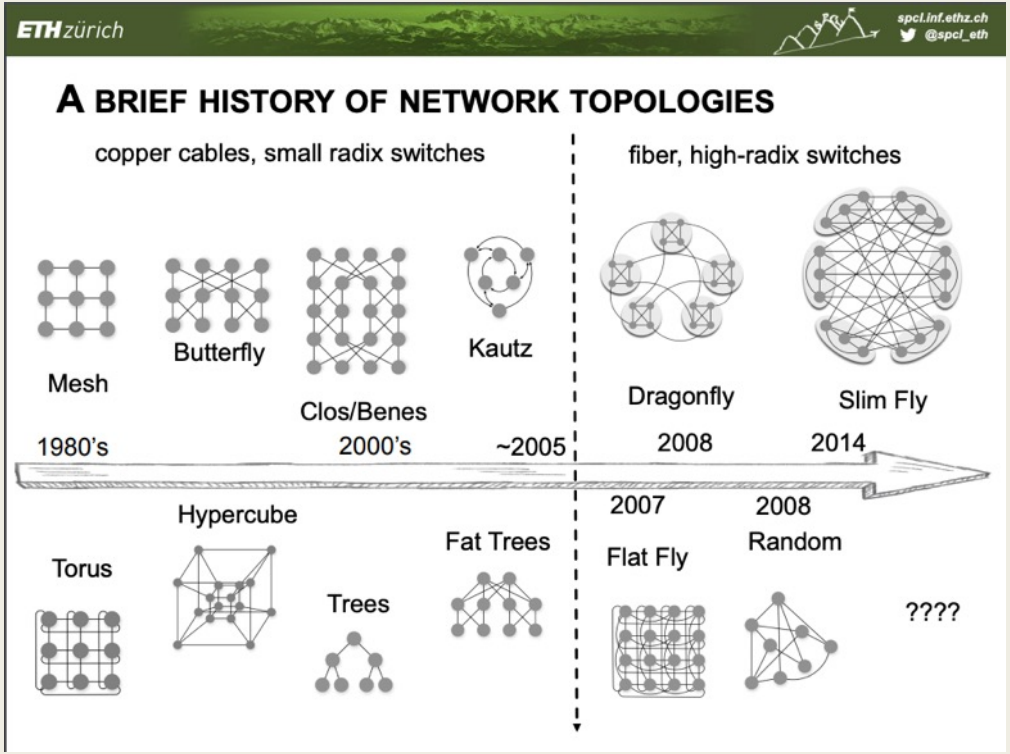


Crossbar and omega interconnect for distributed memory

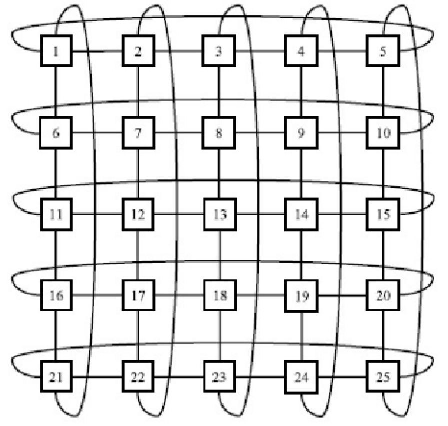




More details on history of network topology from Microsoft Research



https://www.youtube.com/watch?v=F8F0JN6X0fE



How do interconnects and network topology affects performance?

- What are the pro & cons of a torus VS a dragonfly network?
  - Torus allows to communicate quickly with neighbor nodes, but communications
    with nodes far away are slower (non uniform communication patterns)
  - Dragonfly allows to communicate with any node with a small limited number of hops,
    but the dynamic routing is expensive and can cause unexpected performance degradation