Computer Architecture
Parallel Computers

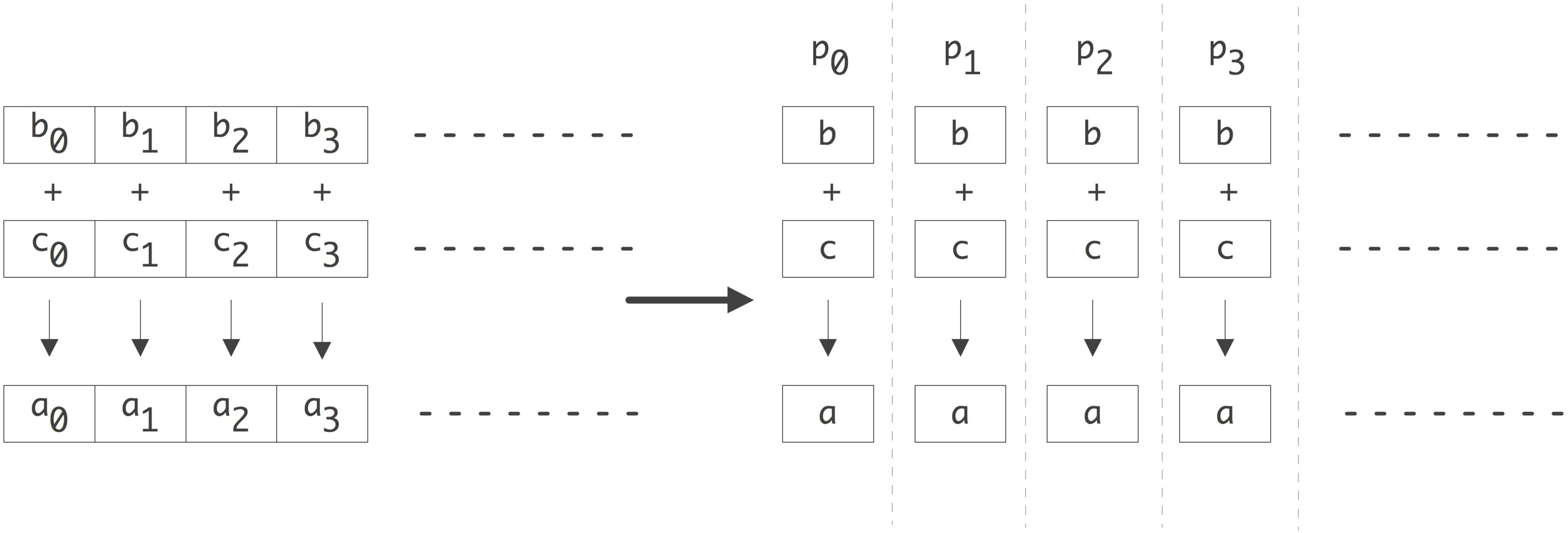
The basic idea

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor,
- Does this become *t/p* on *p* processors (*p<=n*)?

```
for (i=0; i<n; i++)
  a[i] = b[i]+c[i]
```

Idealized version: every process has one array element

```
a = b+c
```


b_0 b_1 b_2 b_3
+ + + +
c_0 c_1 c_2 c_3
a_0 a_1 a_2 a_3
p_0 p_1 p_2 p_3
b b b b
+ + + +
c c c c
a a a a

The basic idea

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor,
- Does this become *t/p* on *p* processors (*p*<=*n*)?

```
for (i=0; i<n; i++)
  a[i] = b[i]+c[i]
```

```
a = b+c
```

Idealized version: every process has one array element

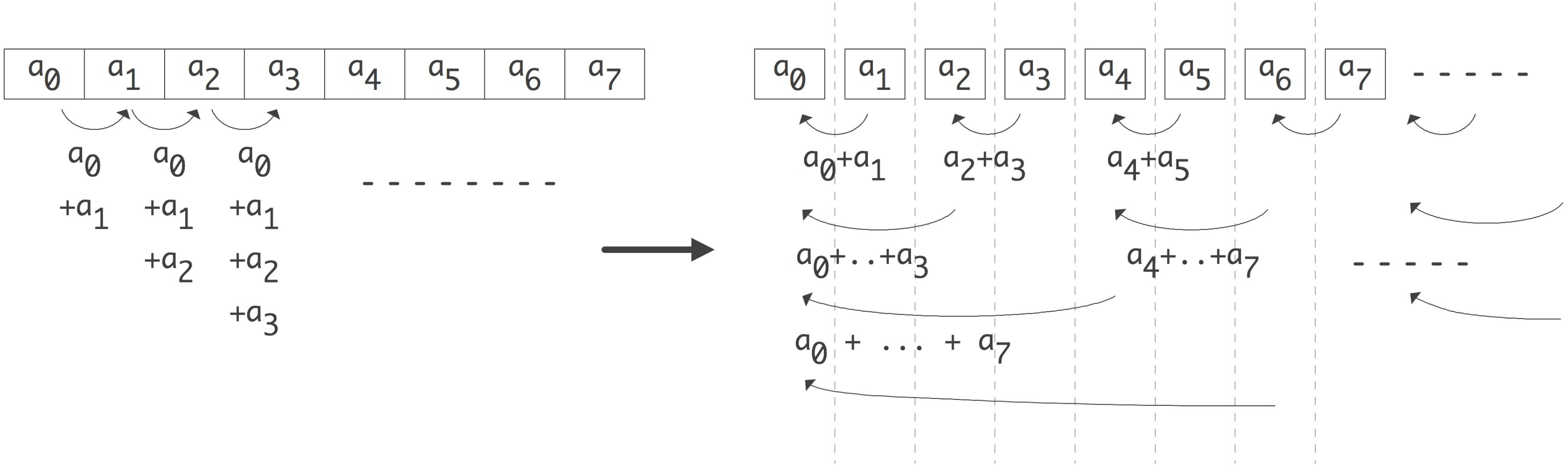
```
for (i=my_low; i<my_high; i++)
  a[i] = b[i]+c[i]
```

Slightly less ideal: each processor has part of the array

The basic idea (cont’d)

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor,
- Does it always become *t/p* on *p* processors (*p*<=*n*)?

```
s = sum( a[i], i=0,n-1 )
```



a0 a1 a2 a3 a4 a5 a6 a7
a0 +a1
a0 +a1 +a2
a0 +a1 +a2 +a3
----
a0 a1 a2 a3 a4 a5 a6 a7 -----
a0+a1 a2+a3 a4+a5
a0+..+a3 a4+..+a7 -----
a0 + ... + a7

The basic idea (cont’d)

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor,
- Does it always become *t/p* on *p* processors (*p<=n*)?

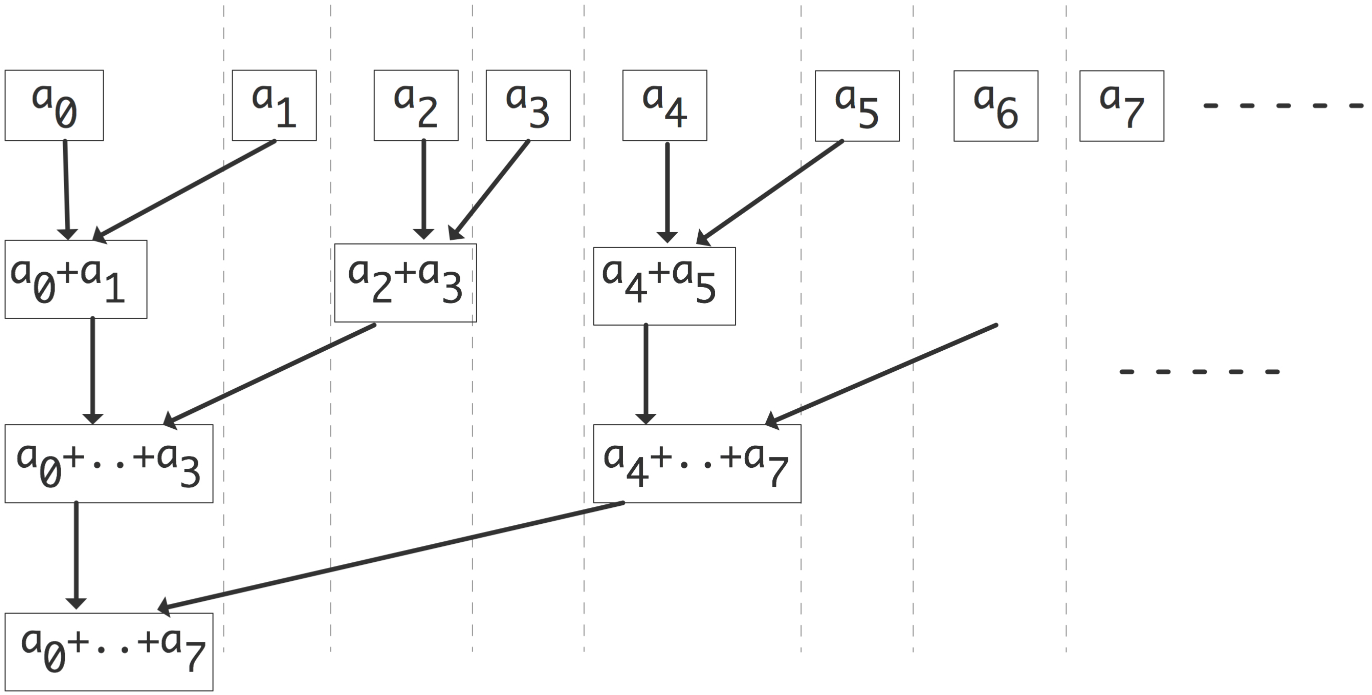
```
s = sum( a[i], i=0,n-1 )
```

```
for (s=2; s<n; s*=2)
  for (i=0; i<n; i+=s)
    a[i] += a[i+s/2]
```

Conclusion: n operations can be done with n/2 processors, in total time $\log_2 n$

Theoretical question: can addition be done faster?

Practical question: can we even do this?


a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
- - - - -
a_0+a_1
a_2+a_3
a_4+a_5
- - - - -
$a_0+..+a_3$
$a_4+..+a_7$
$a_0+..+a_7$

Some theory

- ....before we get into the hardware
- Optimally, *p* processes give $T_P = T_1/p$
- Speedup $S_P = T_1/T_p$, is *p* at best
- Superlinear speedup not possible in theory, sometimes happens in practice.
- Perfect speedup in “embarrassingly parallel applications”
- Less than optimal: overhead, sequential parts, dependencies

Some more theory

- ….before we get into the hardware
- Optimally, p processes give $T_P=T_1/p$
- Speedup $S_P = T_1/T_p$, is p at best
- Efficiency $E_P = S_p/p$
- Scalability: efficiency bounded below

Scaling

- Increasing the number of processors for a given problem makes sense up to a point: *p > n/2* in the addition example has no use
- **Strong scaling**: problem constant, number of processors increasing
- More realistic: scaling up problem and processors simultaneously, for instance to keep data per processor constant: **Weak scaling**
- Weak scaling not always possible: problem size depends on measurements or other external factors.

Amdahl’s Law

- Some parts of a code are not parallelizable
- => they ultimately become a bottleneck
- For instance, if 5% is sequential, you can not get a speedup over 20, no matter *p*.
- Formally, if F_s is the sequential fraction and F_p the parallelizable fraction (F_p+F_s=1):
  - T_p= *(sequential) + (parallelized)* = $(T_1F_s) + (T_1F_p/p)$
- *Amdahl’s law:* $\boldsymbol{T_p=T_1(F_s+F_p/p)}$
  - T_p approaches (T_1F_s) as p increases; speedup $S_p <= 1/F_s$

Theoretical characterization of architectures

Parallel Computers Architectures

- **Parallel computing** means using multiple processors, possibly comprising multiple computers
- Flynn's (1966) taxonomy is a first way to classify parallel computers into one of four types:
  - (**SISD**) Single instruction, single data
    - Your (old, single core) desktop
  - (**SIMD**) Single instruction, multiple data
    - Thinking machines CM-2, Cray 1, and other vector machines (there’s some controversy here)
    - Parts of modern GPUs
  - (**MISD**) Multiple instruction, single data
    - Special purpose machines
    - No commercial, general purpose machines
  - (**MIMD**) Multiple instruction, multiple data
    - Nearly all of today’s parallel machines, including your laptop

SIMD

- Based on regularity of computation: all processors often doing the same operation: *data parallel*
- Big advantage: processor do not need separate ALU
- => lots of small processors packed together
- Ex: Goodyear MPP: 64k processors in 1983
- Use masks to let processors differentiate

SIMD then and now

- There used to be computers that were entirely SIMD (usually attached processor to a front end)
- SIMD these days:
  - SSE instructions in regular CPUs
  - GPUs are SIMD units (sort of)

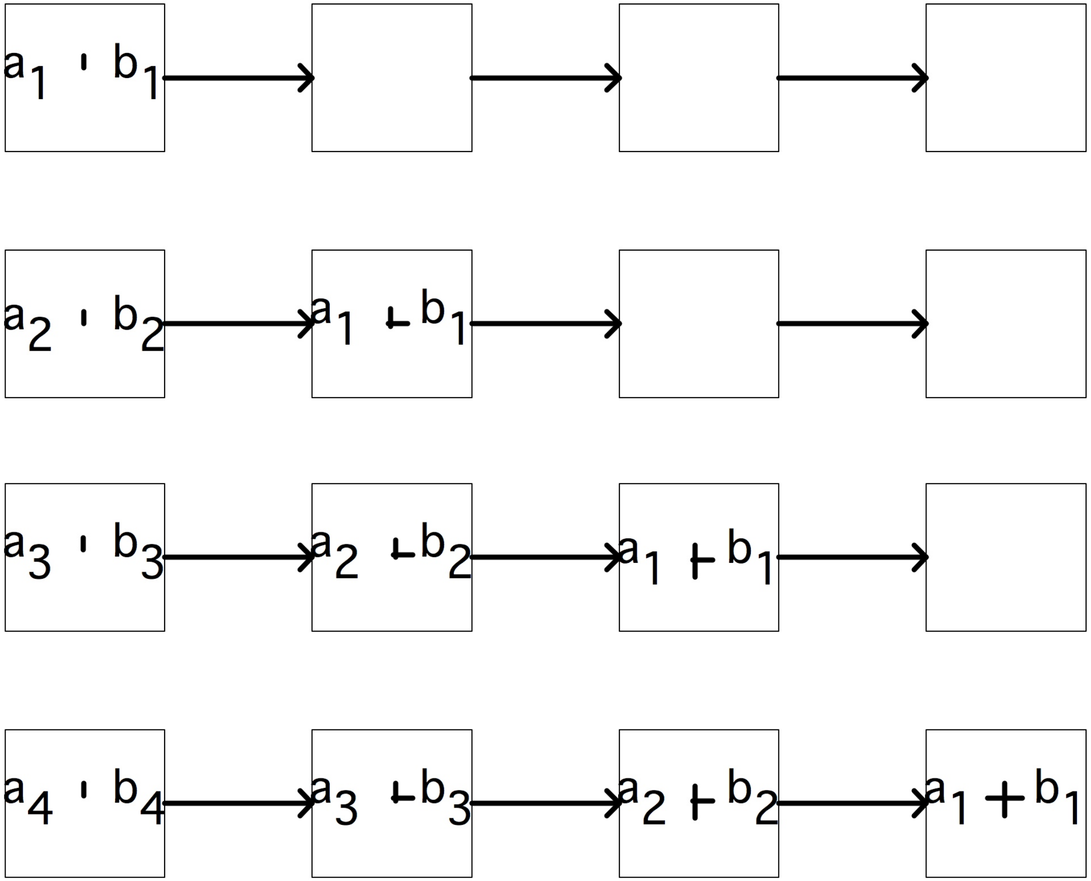
Kinda SIMD: Vector Machines

- Based on a single processor with:
  - Segmented (pipeline) functional units
  - Needs sequence of the same operation
- Dominated early parallel market
  - overtaken in the 90s by clusters, et al.
- Making a comeback (sort of)
  - clusters/constellations of vector machines:
    - Earth Simulator (NEC SX6) and Cray X1/X1E
  - Arithmetic units in CPUs are pipelined.

Remember the pipeline

- Assembly line model (body on frame, attach wheels, doors, handles on doors)
- Floating point addition: exponent align, add mantissas, exponent normalize
- Separate hardware for each stage: pipeline processor

$c_i \leftarrow a_i + b_i$



$a_1 + b_1$			
$a_2 + b_2$	$a_1 + b_1$		
$a_3 + b_3$	$a_2 + b_2$	$a_1 + b_1$	
$a_4 + b_4$	$a_3 + b_3$	$a_2 + b_2$	$a_1 + b_1$

MIMD

- Multiple Instruction, Multiple Data
- Most general model: each processor works on its own data with its own data stream: *task parallel*
- Example: one processor produces data, next processor consumes/analyzes data

MIMD

- In practice SPMD: Single Program Multiple Data:
  - all processors execute the same code
  - Just not the same instruction at the same time
  - Different control flow possible too
  - Different amounts of data: load unbalance

Granularity

- You saw data parallel and task parallel
- Medium grain parallelism: carve up large job into tasks of data parallel work
- (Example: array summing, each processor has a subarray)
- Good match to hybrid architectures:

  task -> node
  data parallel -> SIMD engine

GPU: the miracle architecture (?)

- Lots of hype about incredible speedup / high performance for low cost. What's behind it?
- Origin of GPUs: that "G"
- Graphics processing: identical (fairly simple) operations on lots of pixels
- Doesn't matter when any individual pixel gets processed, as long as they all get done in the end
- (Otoh, CPU: heterogeneous instructions, need to be done ASAP.)
- => GPU is SIMD engine
- …and scientific computing is often very data-parallel

GPU programming:

- **`KernelProc<< m,n >>( args )`**
- Explicit SIMD programming
- There is more: threads (see later)

Characterization by Memory structure

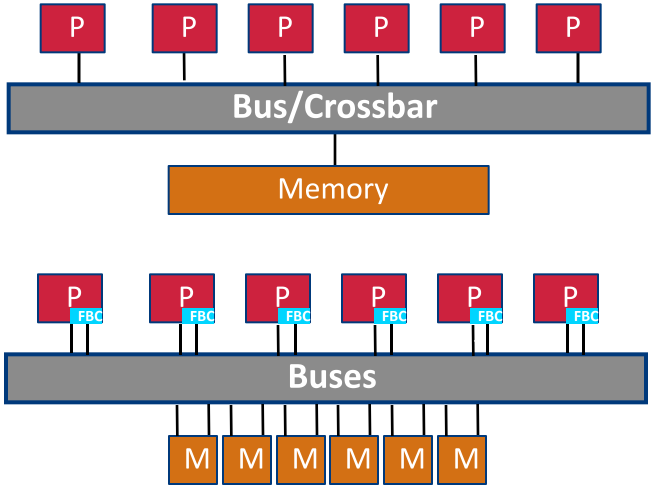
Parallel Computer Architectures

- Top500 List now dominated by MPPs and Clusters
- The MIMD model "won".
- SIMD exists only on smaller scale
- A much more useful way to classification is by memory model
  - **shared** memory
  - **distributed** memory

Two memory models

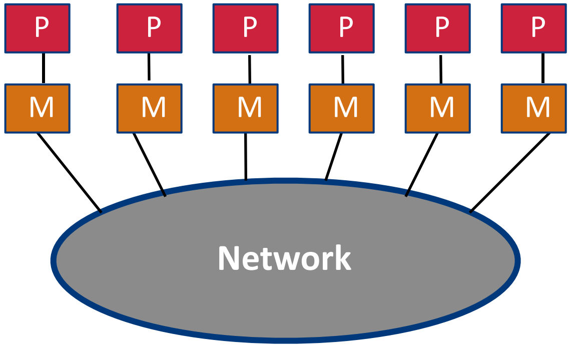
- Shared memory: all processors share the same address space
  - **OpenMP**: directives-based programming
  - PGAS languages (UPC, Titanium, X10)
- Distributed memory: every processor has its own address space
  - **MPI**: Message Passing Interface

Shared and Distributed Memory



Shared memory: single address space. All processors have access to a pool of shared memory.
(e.g., Single Cluster node (2-way, 4-way, ...))

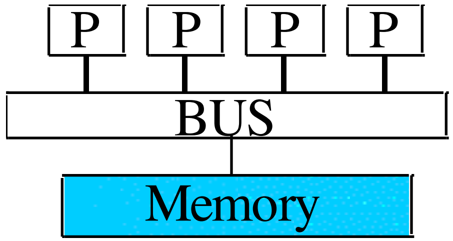
Methods of memory access :
- Bus
- Distributed Switch
- Crossbar



Distributed memory: each processor has its own local memory. Must do message passing to exchange data between processors.
(examples: Linux Clusters, Cray XT3)

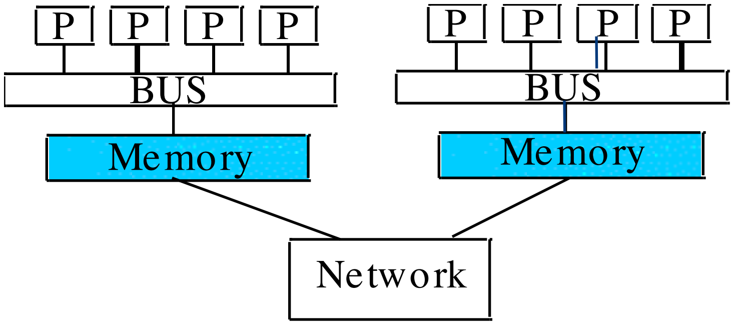
Methods of memory access :
- single switch or switch hierarchy with fat tree, etc. topology

Shared Memory: UMA and NUMA



Uniform Memory Access (UMA): Each processor has uniform access time to memory - also known as symmetric multiprocessors (SMPs) (example: Sun E25000 at TACC)

Non-Uniform Memory Access (NUMA): Time for memory access depends on location of data; also known as Distributed Shared memory machines. Local access is faster than non-local access. Easier to scale than SMPs (e.g.: SGI Origin 2000)



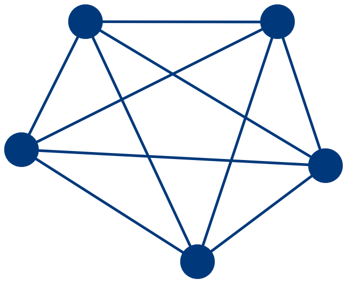
Interconnects

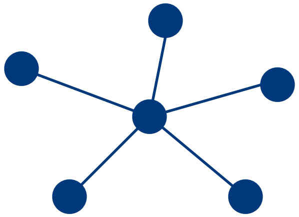
Topology of interconnects

- What is the actual ‘shape’ of the interconnect? Are the nodes connected by a 2D mesh? A ring? Something more elaborate?
- => some graph theory

Completely Connected and Star Networks

- Completely Connected : Each processor has direct communication link to every other processor

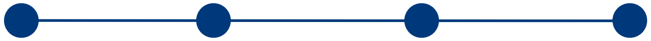




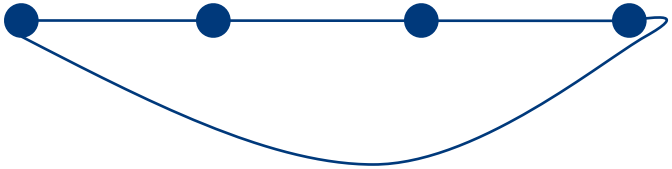
- Star Connected Network : The middle processor is the central processor; every other processor is connected to it.

Arrays and Rings

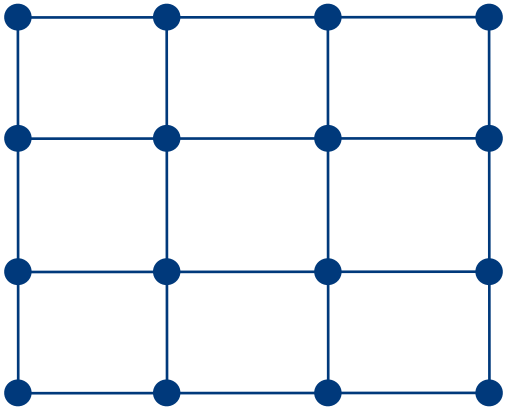
- Linear Array :



- Ring :

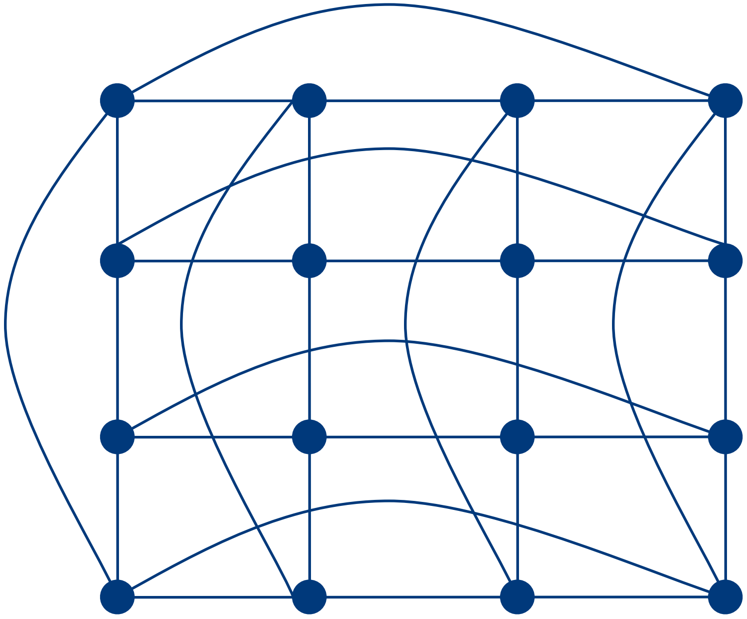


- Mesh Network (e.g. 2D-array)



Torus

2-d Torus (2-d version of the ring)

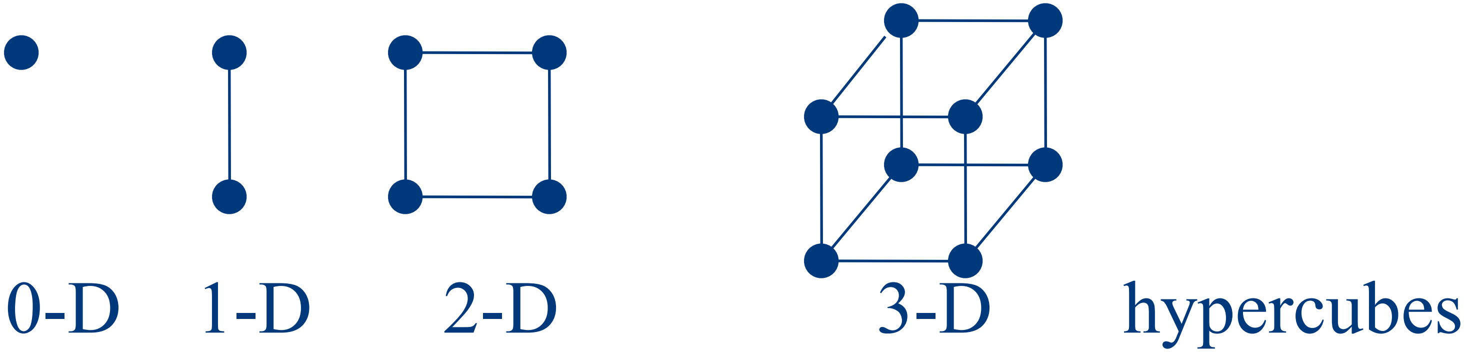


Hypercubes

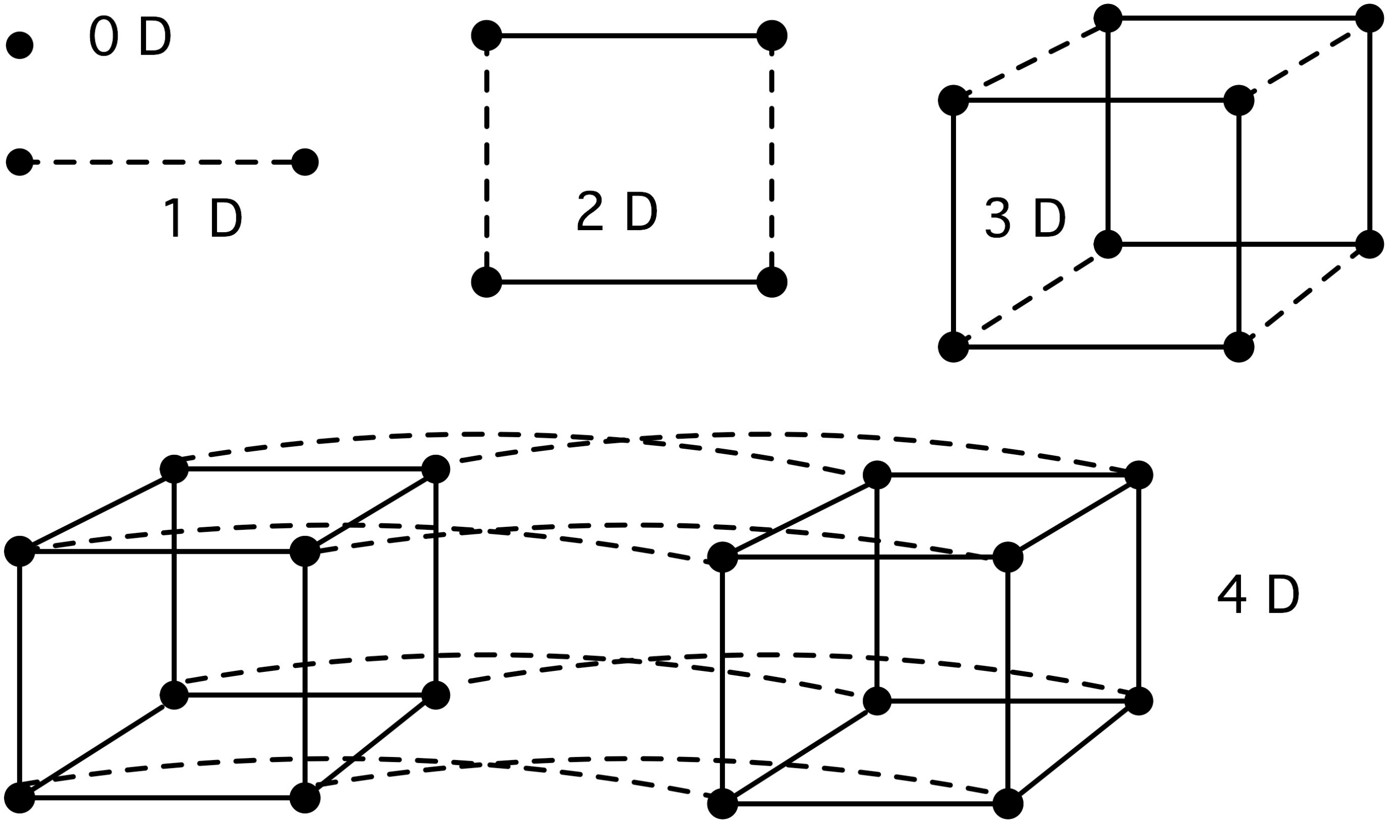
- Hypercube Network : A multidimensional mesh of processors with exactly two processors in each dimension. A d dimensional processor consists of

$$p = 2^d \text{ processors}$$

- Shown below are 0, 1, 2, and 3D hypercubes



Inductive definition

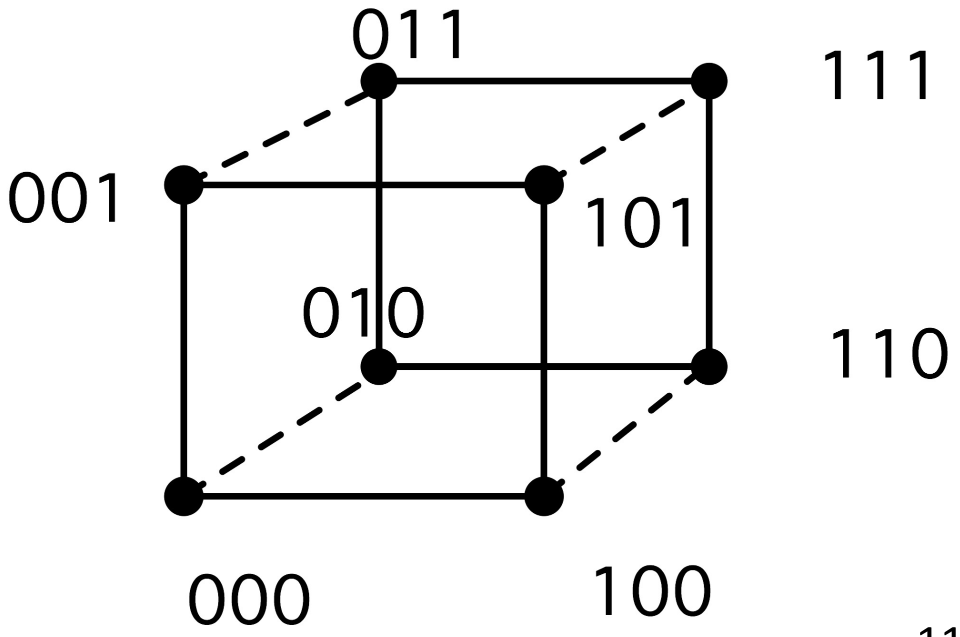


Pros and cons of hypercubes

- Pro: processors are close together: never more than log(p)
- Lots of bandwidth
- Little chance of contention
- Con: the number of wires out of a processor depends on p: complicated design
- Values of p other than 2^p not possible.

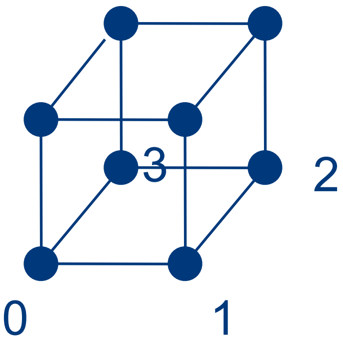
Mapping applications to hypercubes

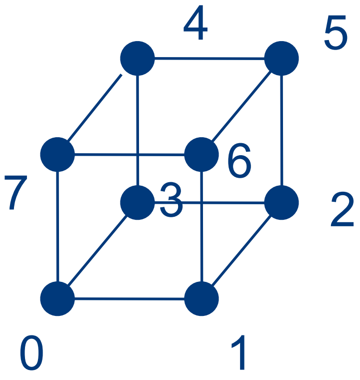
- Is there a natural mapping from 1,2,3D to a hypercube?
- Naïve node numbering does not work:
- Nodes 0 and 1 have distance 1, but
- 3 and 4 have distance 3
- (so do 7 and 0)



Mapping applications to hypercubes

- Is there a natural mapping from 1,2,3D to a hypercube?
- => Gray codes
- Recursive definition: number subcube, then other subcube in mirroring order.





Subsequent processors (in the Linear ordering) all one link apart

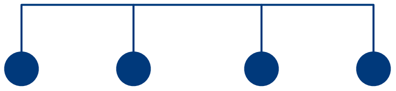
Recursive definition:
0 | 1

0 0 | 1 1
0 1 | 1 0

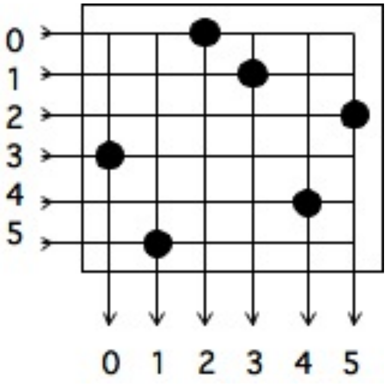
0 0 0 0 | 1 1 1 1
0 0 1 1 | 1 1 0 0
0 1 1 0 | 0 1 1 0

Busses/Hubs and Crossbars

Hub/Bus: Every processor shares the communication links

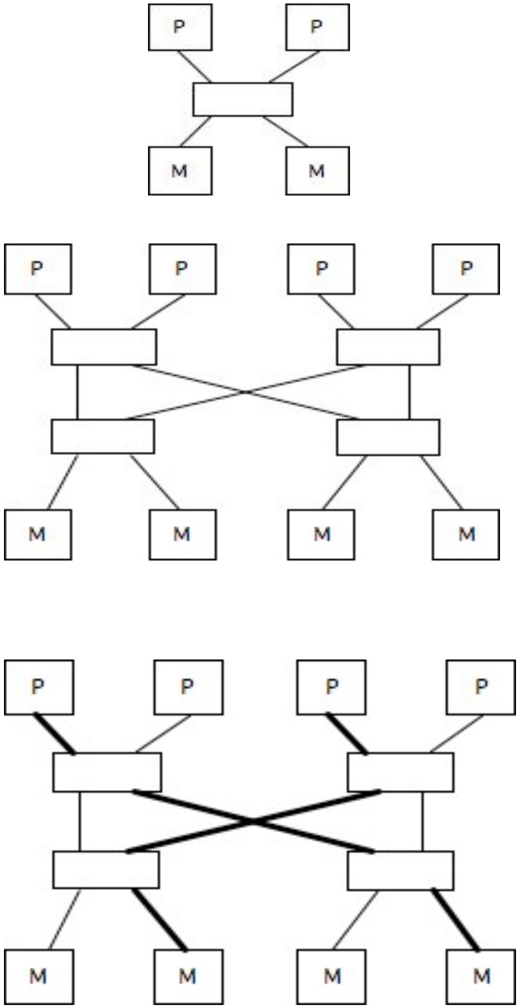


Crossbar Switches: Every processor connects to the switch which routes communications to their destinations



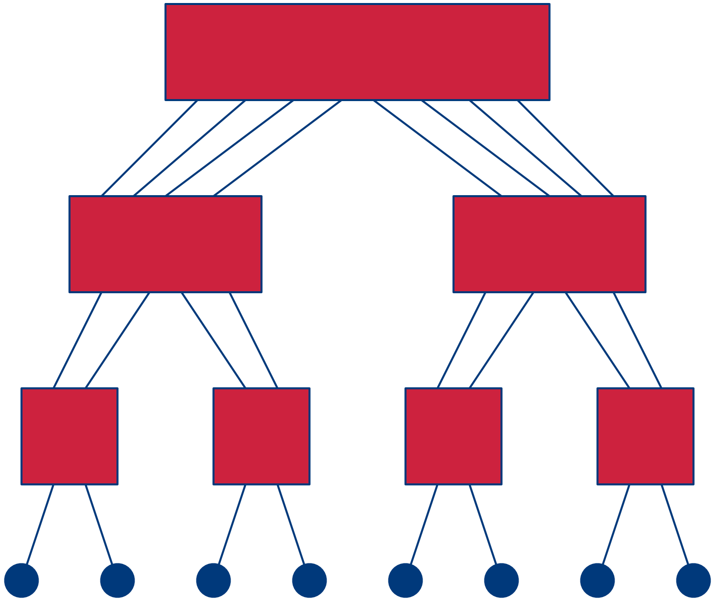
Butterfly exchange network

- Built out of simple switching elements
- Multi-stage; #stages grows with #procs
- Multiple non-colliding paths possible
- Uniform memory access



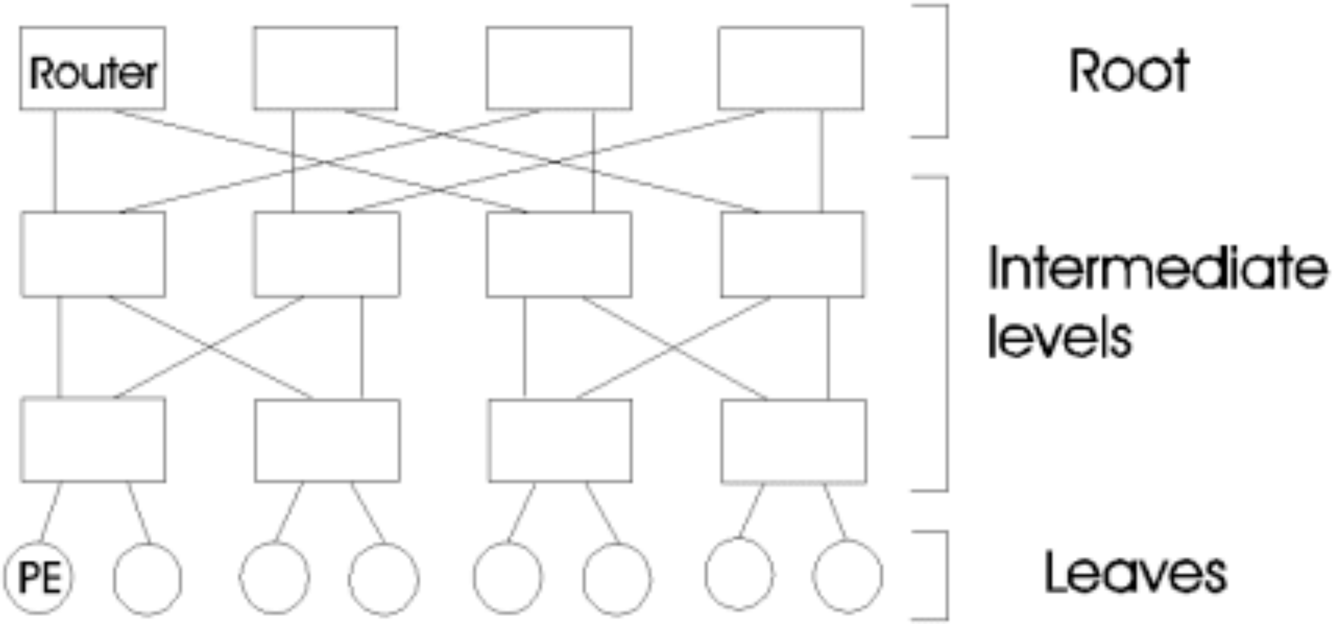
Fat Trees

- Multiple switches
- Each level has the same number of links in as out
- Increasing number of links at each level
- Gives full bandwidth between the links
- Added latency the higher you go



Fat Trees

- In practice emulated by switching network



Interconnect graph theory

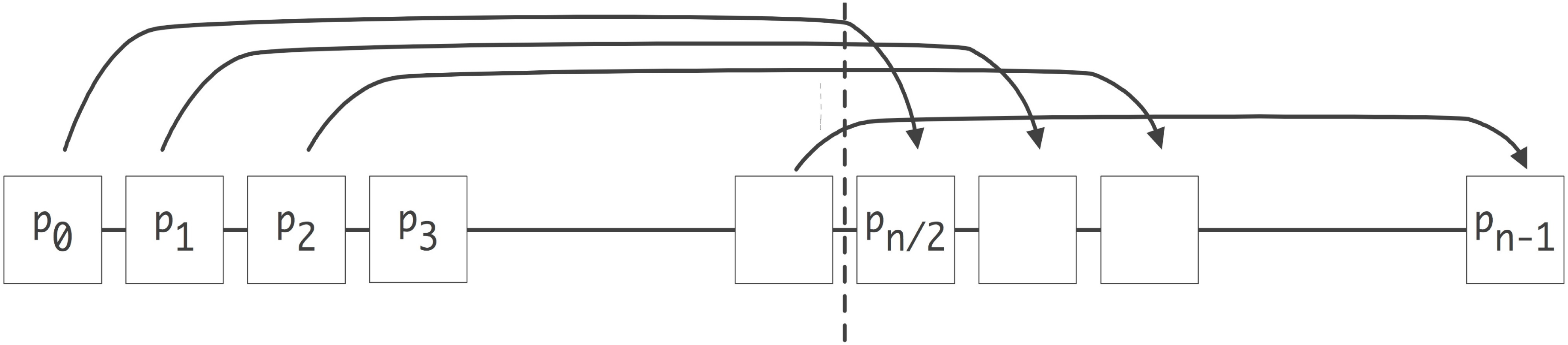
- Degree
  - How many links to other processors does each node have?
  - More is better, but also expensive and hard to engineer
- Diameter
  - maximum distance between any two processors in the network.
  - The distance between two processors is defined as the shortest path, in terms of links, between them.
  - completely connected network is 1, for star network is 2, for ring is p/2 (for p even processors)
- Connectivity
  - measure of the multiplicity of paths between any two processors (# arcs that must be removed to break the connection).
  - high connectivity is desired since it lowers contention for communication resources.
  - 1 for linear array, 1 for star, 2 for ring, 2 for mesh, 4 for torus
  - technically 1 for traditional fat trees, but there is redundancy in the switch infrastructure

Practical issues in interconnects

- Latency : How long does it take to start sending a "message"? Units are generally microseconds or milliseconds.
- Bandwidth : What data rate can be sustained once the message is started? Units are Mbytes/sec or Gbytes/sec.
  - Both point-to-point and aggregate bandwidth are of interest
- Multiple wires: multiple latencies, same bandwidth
- Sometimes shortcuts possible: `wormhole routing’

Measures of bandwidth

- Aggregate bandwidth: total data rate if every processor sending: total capacity of the wires. This can be very high and quite unrealistic.
- Imagine linear array with processor *i* sending to *P/2+i*: `Contention’
- Bisection bandwidth: bandwidth across the minimum number of wires that would split the machine in two.


p0
p1
p2
p3
pn/2
pn-1

Interconnects

- Bisection width
  - Minimum # of communication links that have to be removed to partition the network into two equal halves. Bisection width is
  - 2 for ring, sq. root(p) for mesh with p (even) processors, p/2 for hypercube, (p*p)/4 for completely connected (p even).
- Channel width
  - of physical wires in each communication link
- Channel rate
  - peak rate at which a single physical wire link can deliver bits
- Channel BW
  - peak rate at which data can be communicated between the ends of a communication link
  - = (channel width) * (channel rate)
- Bisection BW
  - minimum volume of communication found between any 2 halves of the network with equal # of procs
  - = (bisection width) * (channel BW)

Bandwidth and Latency

	IB-DDR	10 Gigabit	1 Gigabit
Ping-Pong bandwidth, MB/s	1466	1000	112.5
Exchange bandwidth, MB/s	2659	2073	157.6
Latency, us	2.01	8.23	46.52