Design of Parallel Algorithms

The Architecture of a Parallel Computer

Trends in Microprocessor Architectures

- Microprocessor clock speeds are no longer increasing and have reached a limit of 3-4 Ghz (End of Dennard Scaling)
- Transistor counts still are doubling about every 2 years (Moore's Law), but densities are no longer increasing. (End of Moore's Law)
- As transistor costs increase, performance of computer architectures are now increased by exploiting parallelism
  - Deep pipelines
  - Sophisticated instruction reordering hardware
  - Vector like instruction sets (MMX,SSE, Advanced Vector Extensions (AVX))
  - Novel architectures (GPGPUs, FPGA)
  - Multi-Core

Implicit Parallelism

Pipelining and Vector Execution

- Pipelining overlaps various stages of instruction execution to achieve performance.
  - At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
  - This is akin to an assembly line for manufacture of cars.
- Vector execution is one where the same operation is performed on many different data elements, can be used for highly structured computations
  - Usually compilers perform vectorizing analysis to identify computations that can be performed by vector instructions
  - Very high performance libraries usually require some manual intervention to provide vectorization hints to the compiler

+

Pipelining
Architectural Challenges

- The speed of a pipeline is eventually limited by the slowest stage.
  - For this reason, conventional processors rely on very deep pipelines (20 stage pipelines are common).
- However, in typical program traces, every 5-6th instruction is a conditional jump!
  - Pipelines are fast but have high latency, a 20 stage pipeline will not be able to fill with the correct instructions if the conditional branch depends on a value currently in the pipeline!
  - Branch prediction is used to mitigate this problem
  - The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.
  - There is a limit to how much parallelism can be exploited using pipeline strategies
- Special hardware can make use of dynamic information to perform branch prediction and instruction reordering to keep pipelines full
- Does not require as much work for the compiler to exploit

+

Vector Extensions Architectural Challenges

- Vector Extensions (modern version of superscalar) require much more compiler intervention
  - Compiler must identify streams of computations that are well structured to coordinate computations on vector registers
  - Loop unrolling is a typical approach, basically if loop accesses are independent, then execution of several loop iterations at once can be mapped to vector registers
  - Memory alignment also can be a constraint on loading vector registers
- This requires compile-time knowledge of data-flow in programs
  - Loop unrolling requires knowledge of data dependencies in loop. If one iteration writes a to a memory location accessed by a subsequent iteration, then unrolled computations cannot be loaded into vector registers in advance
  - Data-dependencies may be difficult to determine at compile-time, particularly in languages that allow aliasing (more than one way to access the same memory location, usually through pointers)
- Compiler directed vectorization becomes less effective as vector register sizes get larger (harder to do accurate data-dependency analysis)

Multicore Architectural Challenges

- One solution to these problems is to develop multicore architectures
- Can automatically exploit task level parallelism from operating systems when multiple processes are running or when running multithreaded applications
- Automatic compilers for multicore architectures exist, but in general do not achieve good utilization. Generally multicore parallelization requires even more robust dependency analysis than vectorizing optimizations require.
- Usually exploiting multicore architectures requires some level of manual parallelization
  - Applications will need to be rewritten to fully exploit this architectural feature
  - Unfortunately, this currently appears to be the best method to gain performance from the increased transistor densities provided by Moore's Law

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Memory System Performance: Bandwidth and Latency

- It is very important to understand the difference between latency and bandwidth.
- Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want immediate response from the hydrant, it is important to reduce latency.
- If you want to fight big fires, you want high bandwidth.

Memory Architecture Components

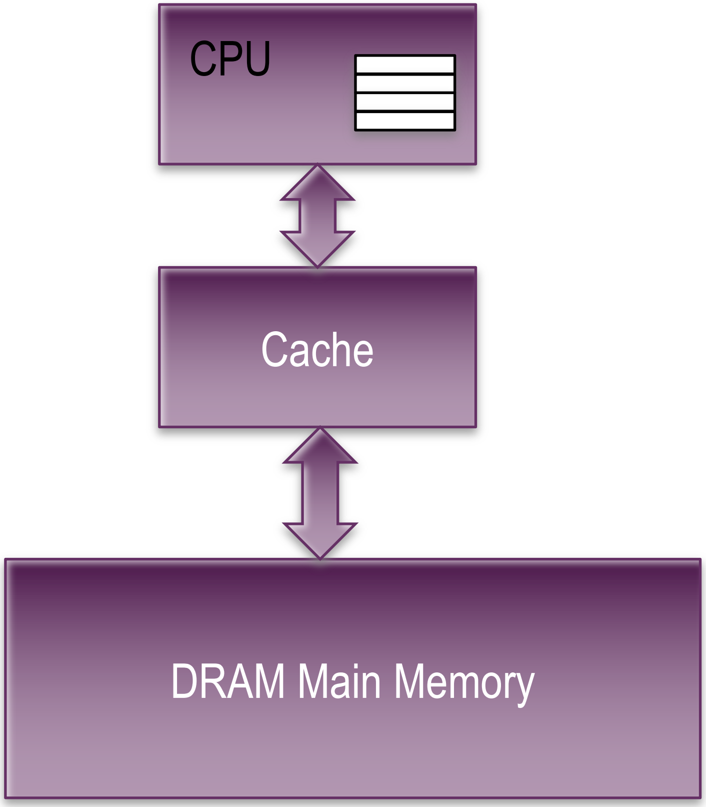
- Static Memory (SRAM)
  - Uses active circuits (consumes power continuously)
  - Large (6 transistors per memory element)
  - High Power (uses power to maintain memory contents)
  - High speed (low latency)
  - Low density
- Dynamic Memory (DRAM) (Must actively refresh to maintain memory)
  - Uses 1 transistor and capacitor per memory element
  - Lower power
  - Slow (high latency)
  - High Density

Design techniques to improve bandwidth and latency in memory systems

- To achieve the required bandwidth we can us parallelism in the memory system
  - Example: If one DRAM chip can access 1 byte every 100ns, then 8 DRAM chips can access 8 bytes every 100ns increasing bandwidth
  - Notice that this technique does not change the latency (access time)
- How do we improve latency? We can't make a 100ns memory go faster than it was designed for…
  - Recognize that for most algorithms there are a set of memory locations that that are accessed frequently called a working set. Use high speed SRAM to store just the working set. This is called a CACHE memory
  - Predict memory accesses and prefetch data before it is needed
  - Use parallelism! If one thread is waiting on memory, switch to other threads that were previously waiting on memory requests. (e.g. hyperthreading)

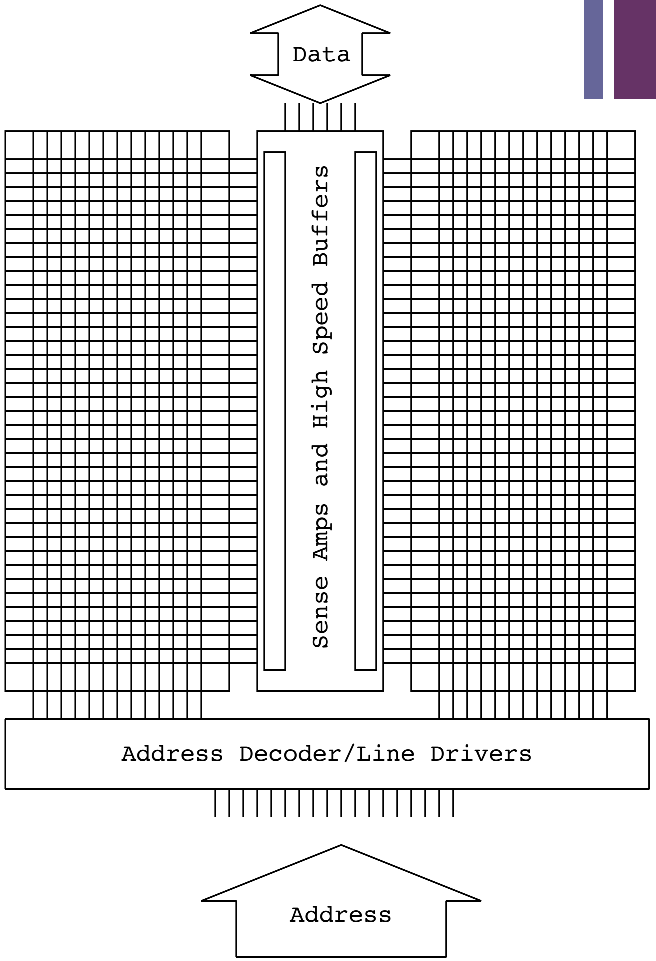
Improving Effective Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
- The fraction of data references satisfied by the cache is called the cache *hit ratio* of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.



DRAM Internal Architecture

- Each memory address request retrieves an entire line which is stored in a fast SRAM buffer.
- Once one word is loaded, then neighboring data can be accessed quickly in a burst mode.
- Since chip pin counts also are a limitation, this design allows more effective utilization of the available pins on a DRAM chip.
- Accessing contiguous segments of memory is highly desirable, not only from a CACHE system perspective, but also from the DRAM architecture itself.


Data
Sense Amps and High Speed Buffers
Address Decoder/Line Drivers
Address

+

Impact of Memory Bandwidth: Example

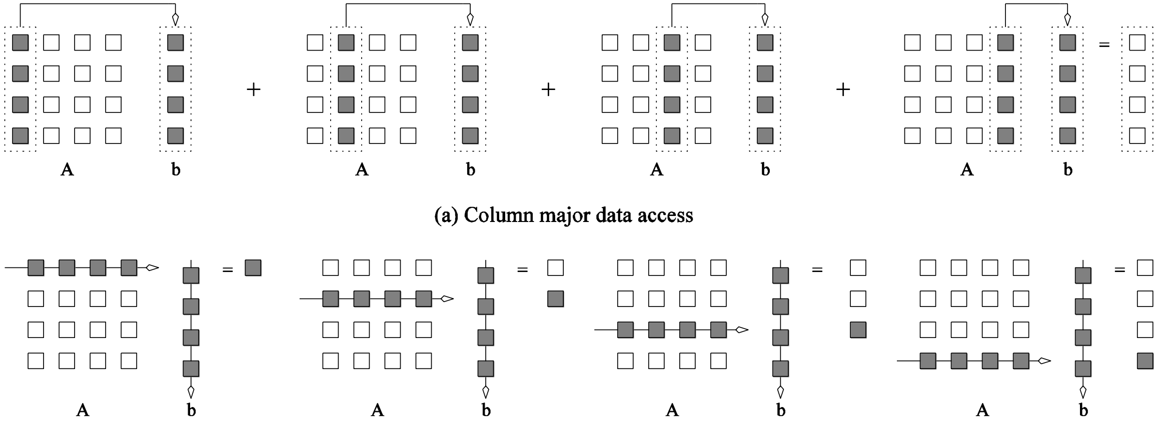
Consider the following code fragment:

```
for (i = 0; i < 1000; i++)
     column_sum[i] = 0.0;
     for (j = 0; j < 1000; j++)
          column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix b into a vector `column_sum`.

Impact of Memory Bandwidth: Example

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order.
- The strided access results in very poor performance.



(a) Column major data access

(b) Row major data access.

Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

Impact of Memory Bandwidth: Example

We can fix the above code as follows:

```
for (i = 0; i < 1000; i++)
      column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
      for (i = 0; i < 1000; i++)
           column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

Memory System Performance: Summary

- The series of examples presented in this section illustrate the following concepts:
  - Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
  - The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
  - Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

+

Explicitly Parallel Platforms

- Parallelism occurs implicitly throughout modern computer designs ranging from pipelines and multiple data paths (vector instructions) in the chip to parallelism in the memory system where many memory chips are accessed simultaneously.
    - This parallelism is managed by the system and compilers and not directly observed by the system programmer
- Parallel clusters and multicore architectures make use of explicit parallelism
    - System programmers are responsible for creating many tasks that can execute simultaneously and be mapped to parallel components of the parallel system by specifying a concurrent control structure.
    - Concurrent tasks must coordinate and share information by way of a communication model.

+

Control Structure of Parallel Programs

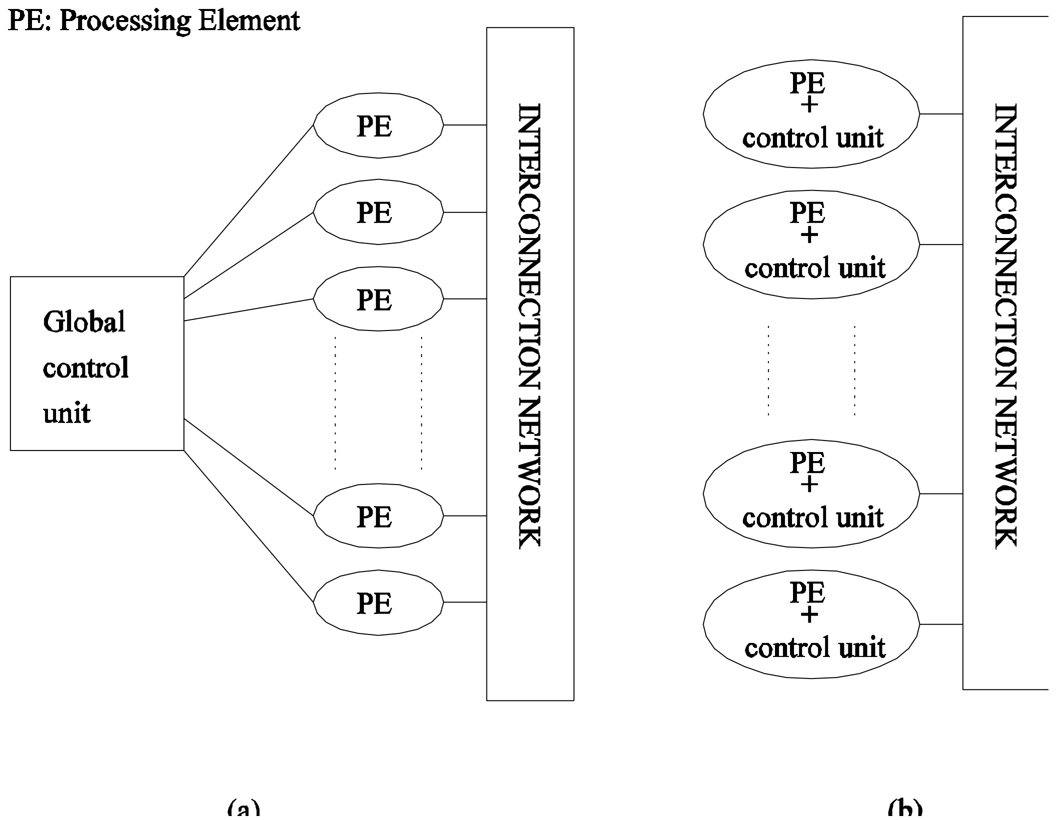
- Parallelism can be expressed at various levels of granularity - from instruction level to processes.
- Between these extremes exist a range of models, along with corresponding architectural support.

Control Structure of Parallel Programs

- Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.
- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).
- If each processor has its own control control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).

+

SIMD and MIMD Processors



A typical SIMD architecture (a) and a typical MIMD architecture (b).

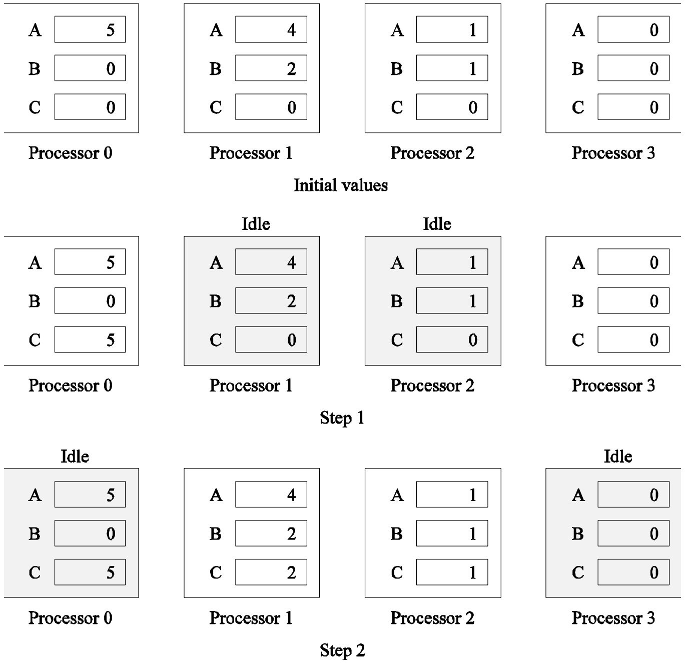
SIMD Processors

- Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines.
- Variants of this concept have found use in co-processing units such as the MMX/SSE/AVX units in Intel processors and DSP chips such as the Sharc.
- SIMD relies on the regular structure of computations (such as those in image processing).
- It is often necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an ``activity mask", which determines if a processor should participate in a computation or not.

+ Conditional Execution in SIMD Processors

```
if (B == 0)
        C = A;
else
        C = A/B;
```

(a)



	Processor 0	Processor 1	Processor 2	Processor 3
A	5	4	1	0
B	0	2	1	0
C	0	0	0	0

Initial values

	Processor 0	Processor 1 Idle	Processor 2 Idle	Processor 3
A	5	4	1	0
B	0	2	1	0
C	5	0	0	0

Step 1

	Processor 0 Idle	Processor 1	Processor 2	Processor 3 Idle
A	5	4	1	0
B	0	2	1	0
C	5	2	1	0

Step 2

(b)

Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

MIMD Processors

- In contrast to SIMD processors, MIMD processors can execute different programs on different processors.
- A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors.
- It is easy to see that SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support.
- Examples of such platforms include almost all modern parallel machines

SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles.
- Not all applications are naturally suited to SIMD processors.
- In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time.

Variant on Themes: Single Instruction Multiple Thread (SIMT)

- In GPGPU architectures the SIMD is generalized somewhat to a paradigm called SIMT (Single Instruction Multiple Threads).
  - In this model threads are dynamically assigned to Warps in which threads that execute the same instruction are grouped together. Warps execute in SIMD fashion.
  - The model is more general than the SIMD model since it is not necessary to idle processors on conditional branching. In this model it is possible to group threads that follow the same path into a single warp avoiding idle processors. For the SIMD model branches have to be dealt with by masking instructions and ignoring them on cores that do not execute them.

Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks - accessing a shared data space and exchanging messages.

- Platforms that provide a shared data space are called shared-address-space machines or multiprocessors.

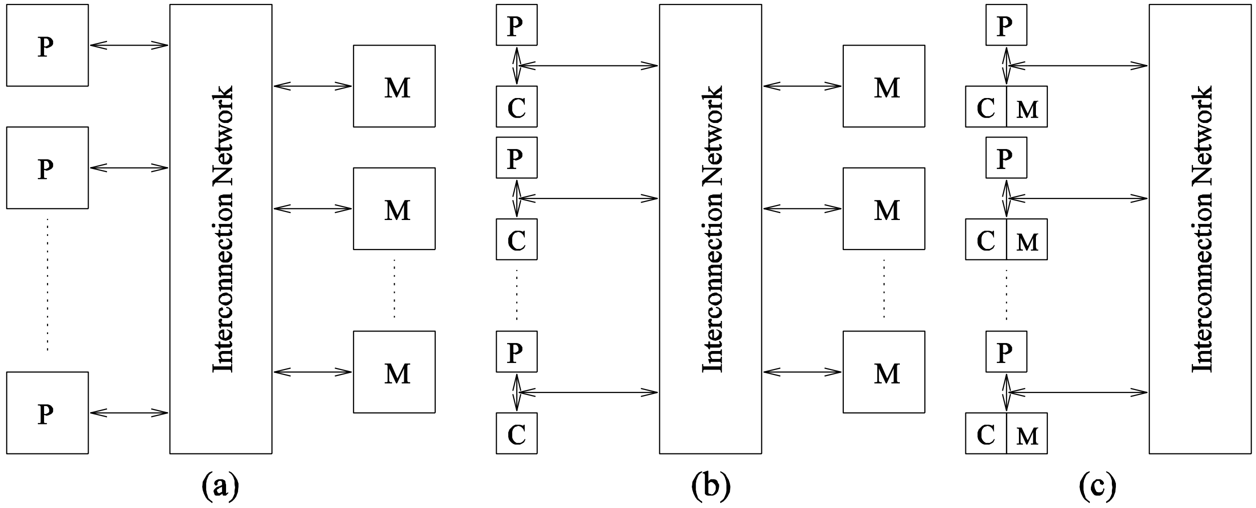
- Platforms that support messaging are also called message passing platforms or multicomputers.

Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system global or local is identical, the platform is classified as a uniform memory access (UMA), else, a non-uniform memory access (NUMA) machine.

+

NUMA and UMA Shared-Address-Space Platforms



Typical shared-address-space architectures: (a) Uniform-memory access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

+

NUMA and UMA Shared-Address-Space Platforms

- The distinction between NUMA and UMA platforms is important from the point of view of algorithm design. NUMA machines require locality from underlying algorithms for performance.
- Programming these platforms is easier since reads and writes are implicitly visible to other processors.
- However, read-write data to shared data must be coordinated (this will be discussed in greater detail when we talk about threads programming).
- Caches in such machines require coordinated access to multiple copies. This leads to the cache coherence problem.
- A weaker model of these machines provides an address map, but not coordinated access. These models are called non cache coherent shared address space machines.

Shared-Address-Space vs. Shared Memory Machines

- It is important to note the difference between the terms shared address space and shared memory.
- We refer to the former as a programming abstraction and to the latter as a physical machine attribute.
- It is possible to provide a shared address space using a physically distributed memory.

+

Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
- Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.
- These platforms are programmed using (variants of) send and receive primitives.
- Libraries such as MPI and PVM provide such primitives.

Message Passing vs. Shared Address Space Platforms

- Message passing requires little hardware support, other than a network.
- Shared address space platforms can easily emulate message passing. The reverse is more difficult to do (in an efficient manner).

Physical Organization of Parallel Platforms

We begin this discussion with an ideal parallel machine called Parallel Random Access Machine, or PRAM.

Natural extension of the RAM architecture which is the traditional serial execution model

- Operations can access memory locations in random order in O(1) time
- Count operations and memory access to model running time

Architecture of an Ideal Parallel Computer

- A natural extension of the Random Access Machine (RAM) serial architecture is the Parallel Random Access Machine, or PRAM.

- This is a theoretical model. Useful for describing parallelization of a program, but many times predicted running times for a PRAM algorithm are highly optimistic.

- PRAMs consist of *p* processors and a global memory of unbounded size that is uniformly accessible to all processors.

- Processors share a common clock but may execute different instructions in each cycle. (synchronization is implicit)

- Programs usually expressed as loops over processors where array addresses are indexed using processor number. These loops are executed in parallel.

Architecture of an Ideal Parallel Computer

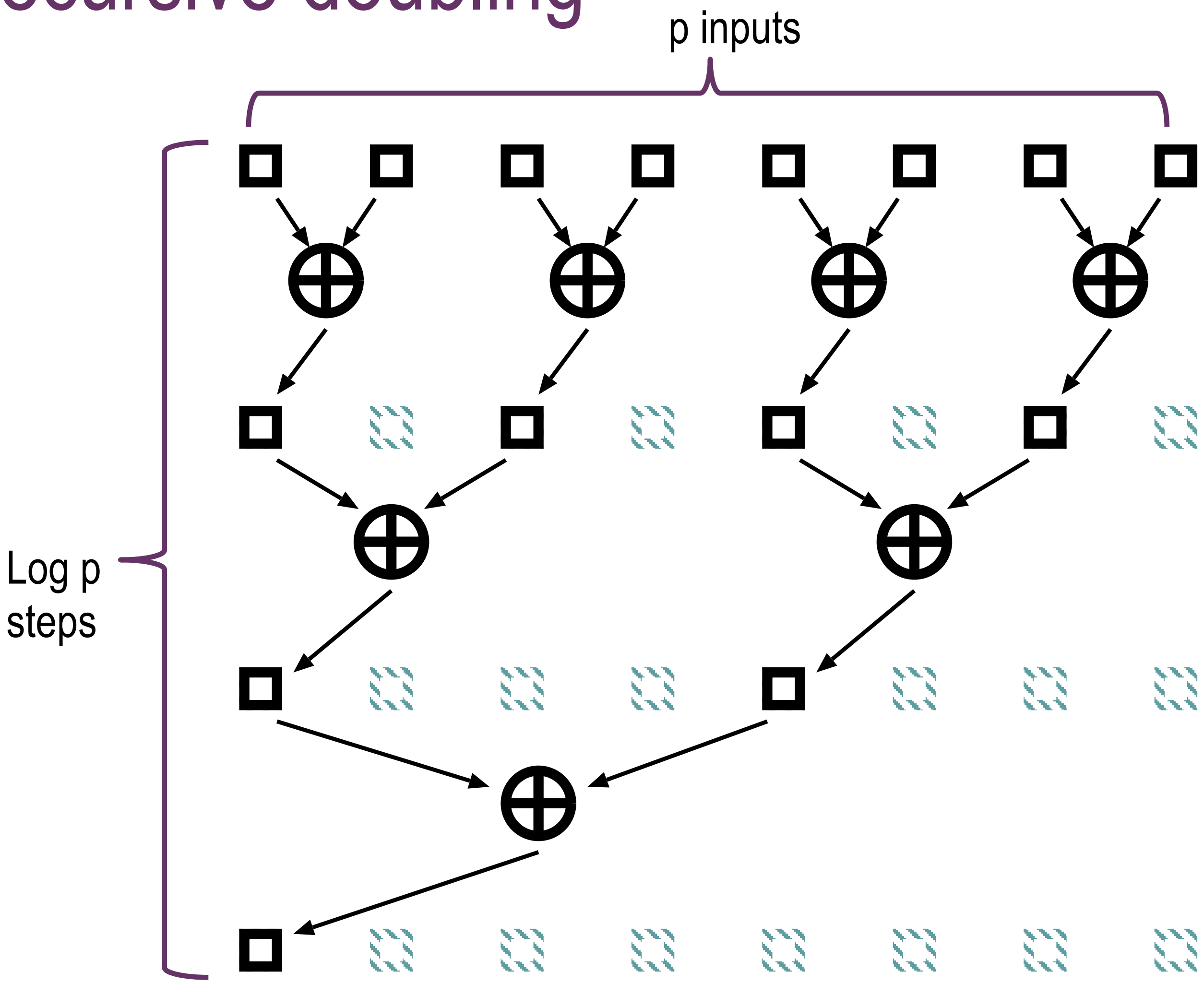
- Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.
  - Exclusive-read, exclusive-write (EREW) PRAM.
  - Concurrent-read, exclusive-write (CREW) PRAM.
  - Exclusive-read, concurrent-write (ERCW) PRAM.
  - Concurrent-read, concurrent-write (CRCW) PRAM.

Architecture of an Ideal Parallel Computer

- Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.
  - Exclusive-read, exclusive-write (EREW) PRAM.
  - Concurrent-read, exclusive-write (CREW) PRAM.
  - Exclusive-read, concurrent-write (ERCW) PRAM.
  - Concurrent-read, concurrent-write (CRCW) PRAM.

- What does concurrent write mean, anyway?
  - Common: write only if all values are identical.
  - Arbitrary: write the data from a randomly selected processor.
  - Priority: follow a predetermined priority order.
  - Sum: Write the sum of all data items.

Summing p numbers in log(p) time using recursive doubling



Example, Summing p numbers with p processors on a EREW PRAM machine

```
int delta = 2
while(delta < p) {
    for each processor i, in parallel
        if(i%delta == 0)
            sum[i] = sum[i] + sum[i+delta]
    delta = delta * 2
}
```

Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as static or dynamic.
- Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

Interconnection Networks

- Switches map a fixed number of inputs to outputs.
- The total number of ports on a switch is the *degree* of the switch.
- The cost of a switch grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.

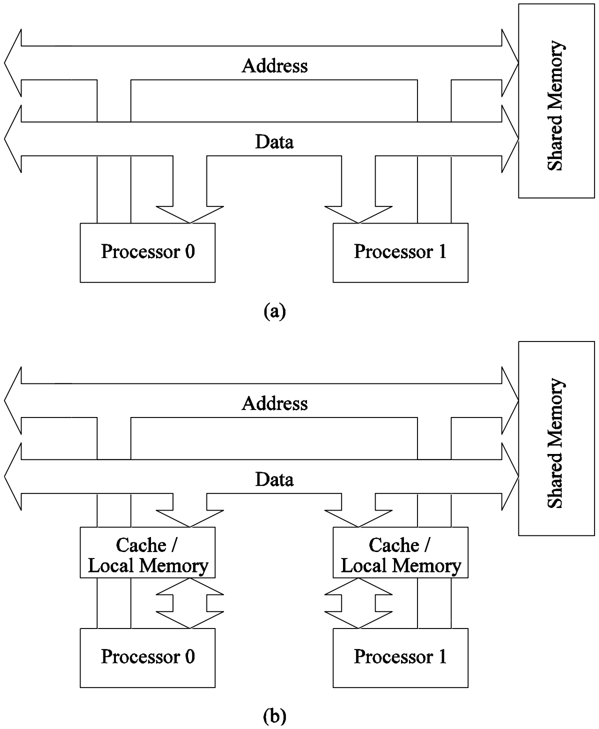
Network Topologies

- A variety of network topologies have been proposed and implemented.
- These topologies tradeoff performance for cost.
- Commercial machines often implement hybrids of multiple topologies for reasons of packaging, cost, and available components.

Network Topologies: Buses

- Some of the simplest and earliest parallel machines used buses.
- All processors access a common bus for exchanging data.
- The distance between any two nodes is *O(1)* in a bus. The bus also provides a convenient broadcast media.
- However, the bandwidth of the shared bus is a major bottleneck.
- Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

Network Topologies: Buses

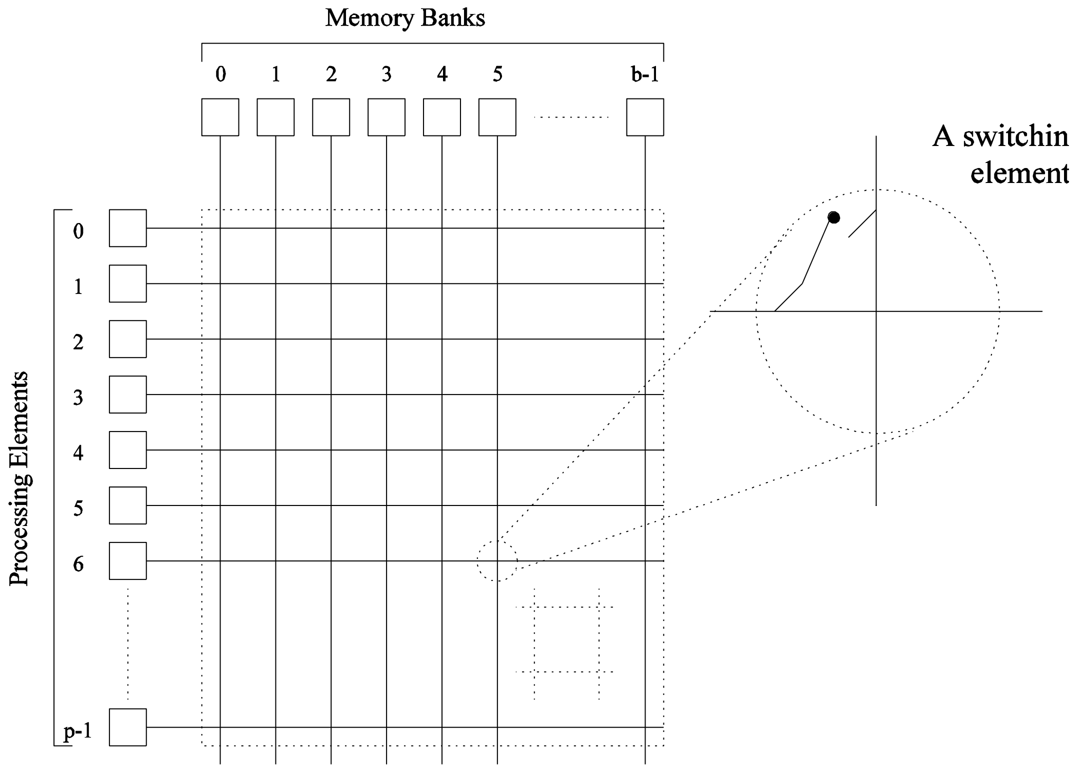


Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.

Network Topologies: Crossbars

A crossbar network uses an *p*×*m* grid of switches to connect *p* inputs to m outputs in a non-blocking manner.



A completely non-blocking crossbar network connecting *p* processors to b memory banks.

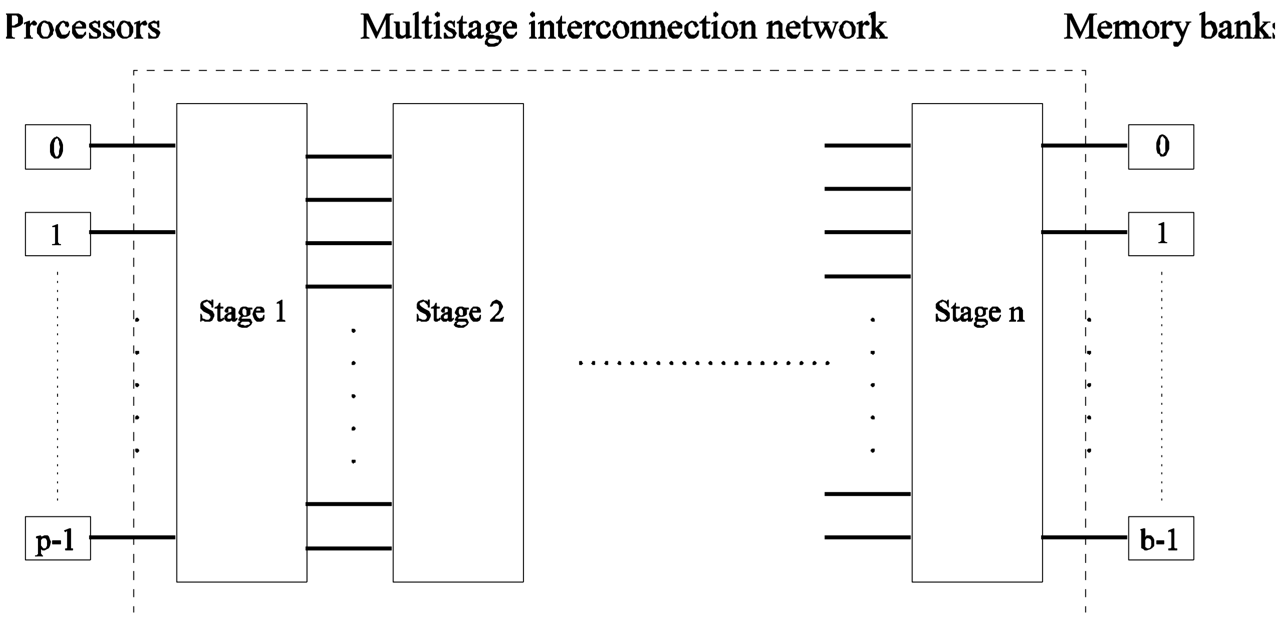
Network Topologies: Crossbars

- The cost of a crossbar of *p* processors grows as $O(p^2)$.
- This is generally difficult to scale for large values of *p*.
- High end shared-memory servers employ cross-bar switches to reduce inter-processor memory latency.
- Not used in modern large scale parallel system due to high cost at large scales

Network Topologies: Multistage Networks

- Crossbars have excellent performance scalability but poor cost scalability.
- Buses have excellent cost scalability, but poor performance scalability.
- Multistage interconnects strike a compromise between these extremes.

Network Topologies: Multistage Networks



The schematic of a typical multistage interconnection network.

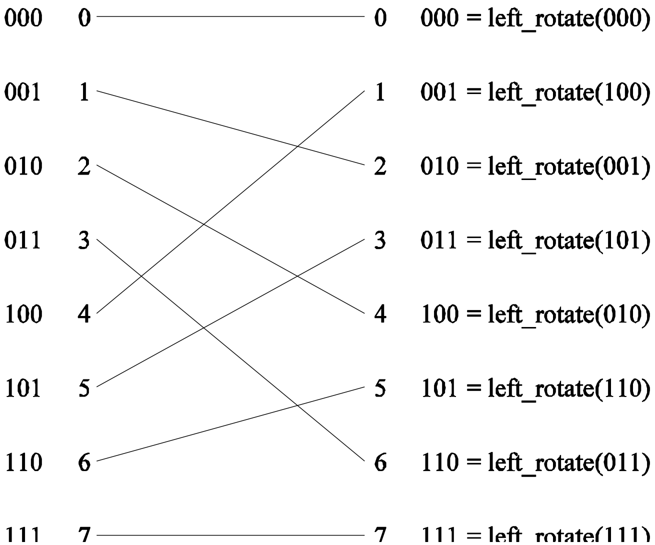
Network Topologies: Multistage Omega Network

- One of the most commonly used multistage interconnects is the Omega network.
- This network consists of *log p* stages, where *p* is the number of inputs/outputs.
- At each stage, input *i* is connected to output *j* if:

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases}$$

Network Topologies: Multistage Omega Network

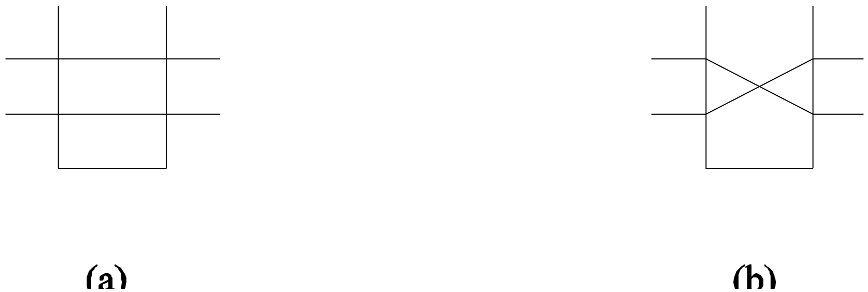
Each stage of the Omega network implements a perfect shuffle as follows:



A perfect shuffle interconnection for eight inputs and outputs.

Network Topologies: Multistage Omega Network

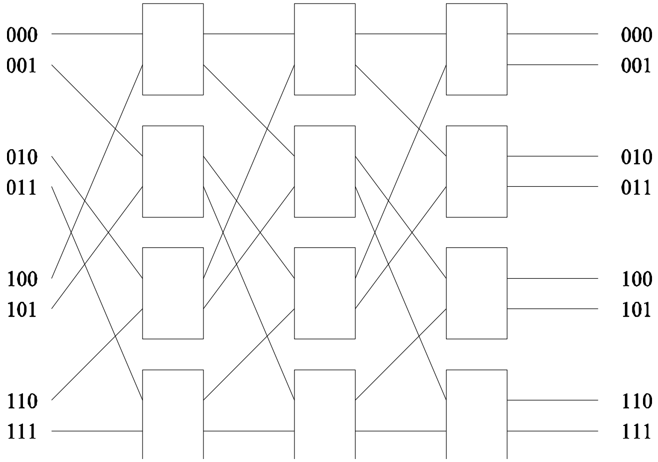
- The perfect shuffle patterns are connected using 2×2 switches.
- The switches operate in two modes – crossover or passthrough.



Two switching configurations of the 2 × 2 switch:
(a) Pass-through; (b) Cross-over.

Network Topologies: Multistage Omega Network

A complete Omega network with the perfect shuffle interconnects and switches can now be illustrated:



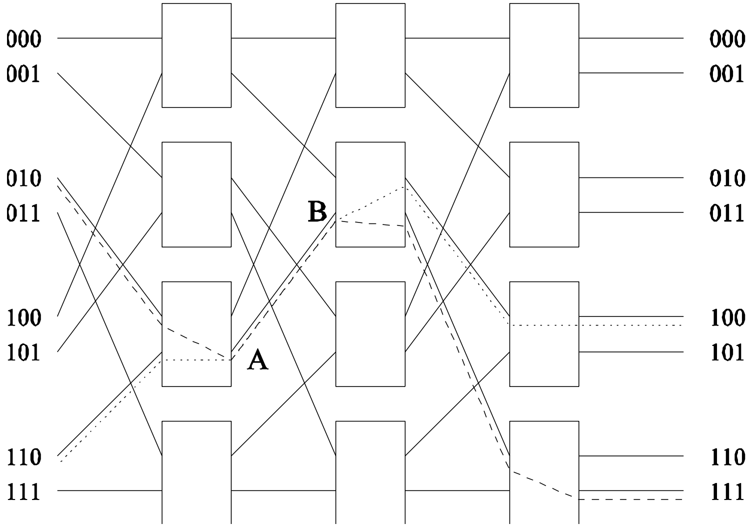
A complete omega network connecting eight inputs and eight outputs.

An omega network has *p/2* × *log p* switching nodes, and the cost of such a network grows as *(p log p).*

Network Topologies: Multistage Omega Network – Routing

- Let *s* be the binary representation of the source and *d* be that of the destination processor.
- The data traverses the link to the first switching node. If the most significant bits of *s* and *d* are the same, then the data is routed in pass-through mode by the switch else, it switches to crossover.
- This process is repeated for each of the *log p* switching stages.
- Note that this is not a non-blocking switch.

Network Topologies:
Multistage Omega Network – Routing



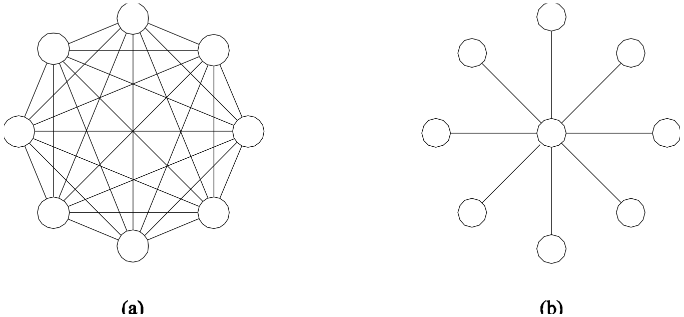
An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

Network Topologies: Completely Connected Network

- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of *p*.
- In this sense, these networks are static counterparts of crossbars.

Network Topologies: Completely Connected and Star Connected Networks

Example of an 8-node completely connected network.



(a) A completely-connected network of eight nodes;
(b) a star connected network of nine nodes.

Network Topologies: Star Connected Network

- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is *O(1).* However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.

Network Topologies: Linear Arrays, Meshes, and *k-d* Meshes

- In a linear array, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a 1-D torus or a ring.

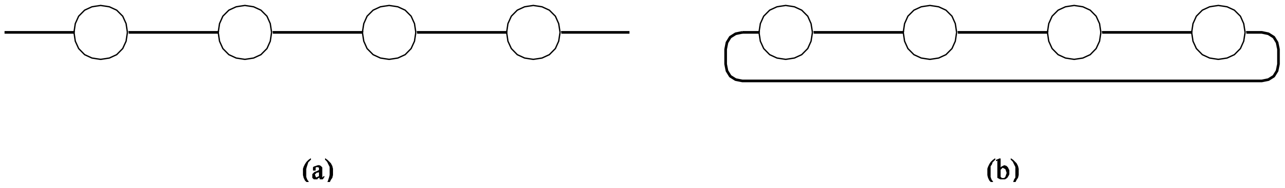
- A generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.

- A further generalization to *d* dimensions has nodes with *2d* neighbors.

- A special case of a *d*-dimensional mesh is a hypercube. Here, $d = \log p$, where *p* is the total number of nodes.

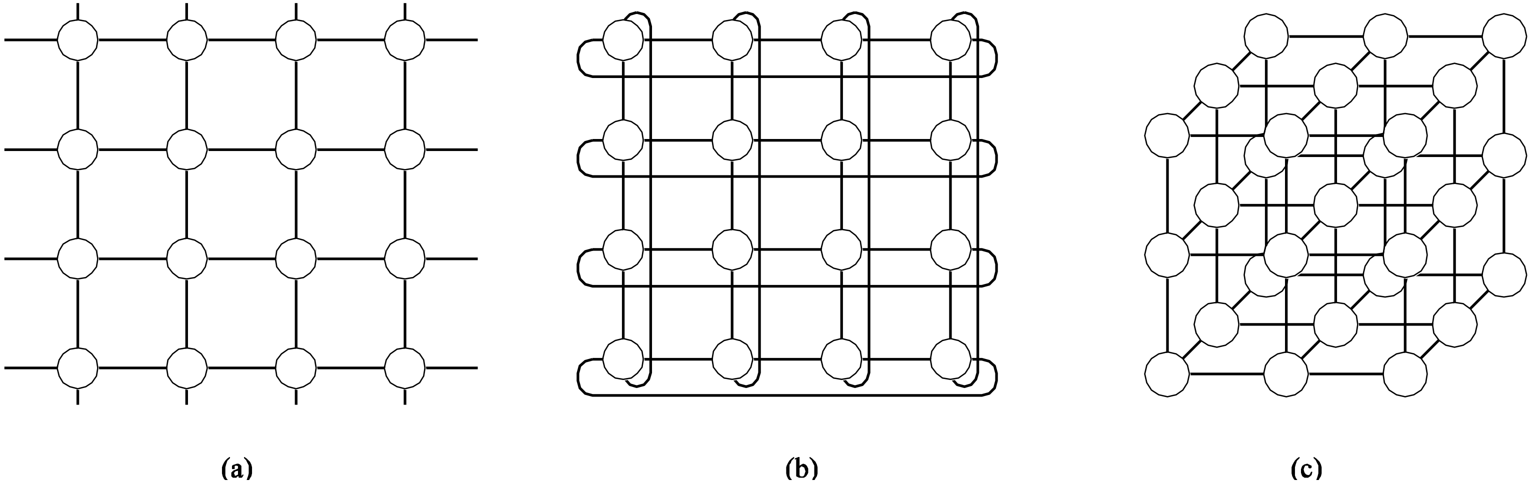
+

Network Topologies: Linear Arrays



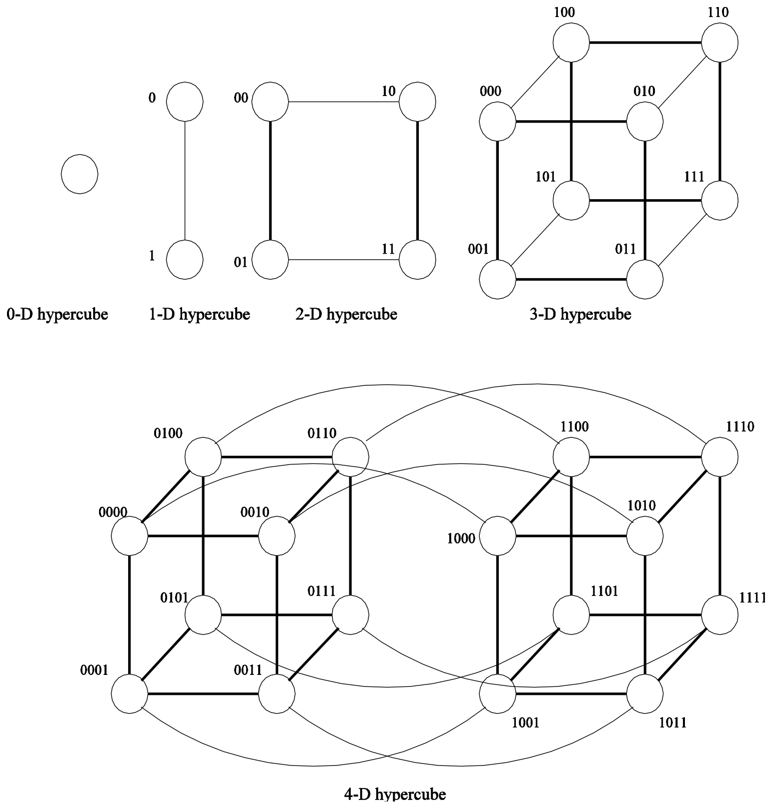
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

Network Topologies: Two- and Three Dimensional Meshes



Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

Network Topologies: Hypercubes and their Construction



0-D hypercube

1-D hypercube

2-D hypercube

3-D hypercube

4-D hypercube

Construction of hypercubes from hypercubes of lower dimension.

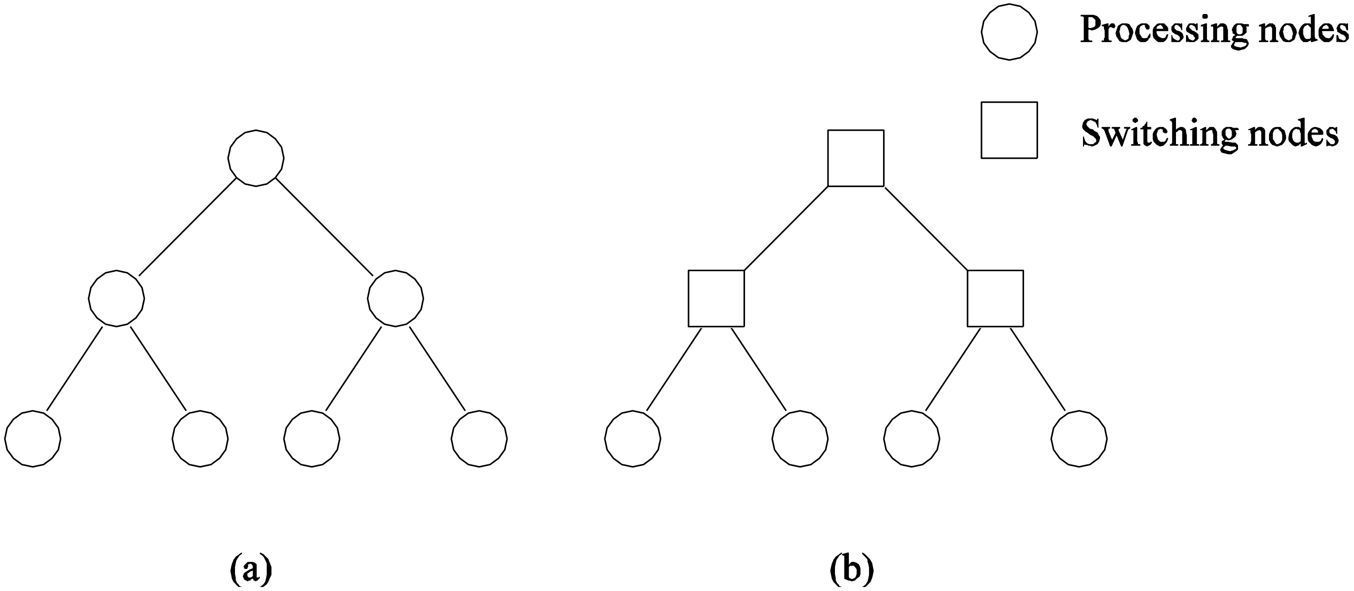
+

Network Topologies: Properties of Hypercubes

- The distance between any two nodes is at most *log p*.
- Each node has *log p* neighbors.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.

+

Network Topologies: Tree-Based Networks

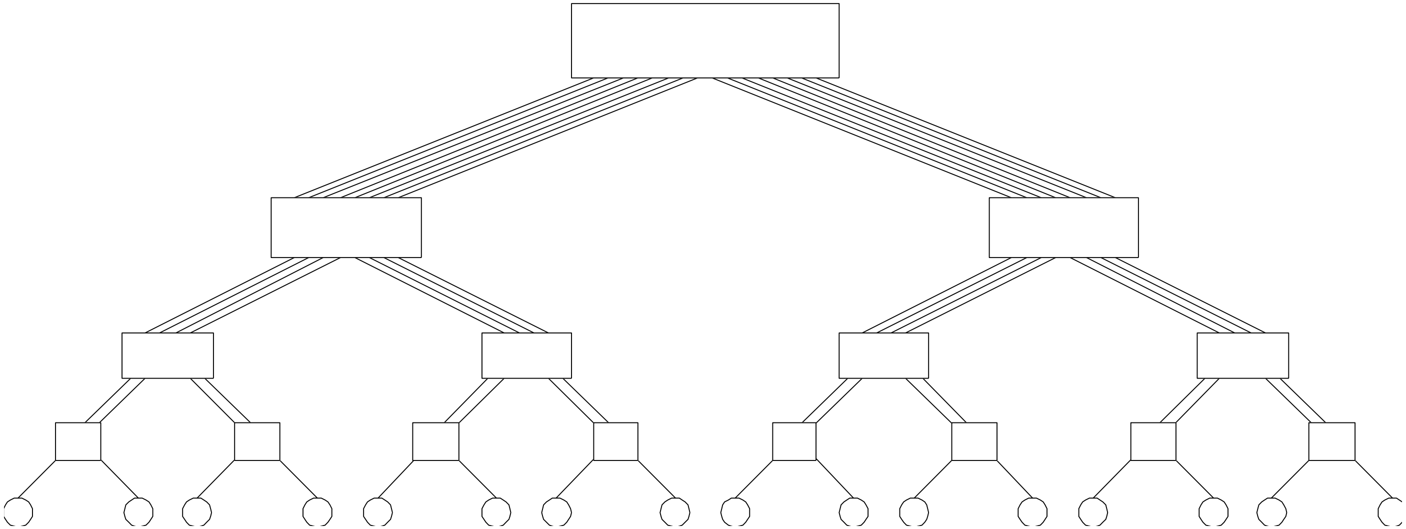


Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

Network Topologies: Tree Properties

- The distance between any two nodes is no more than *2logp*.
- Links higher up the tree potentially carry more traffic than those at the lower levels.
- For this reason, a variant called a fat-tree, fattens the links as we go up the tree.
- Trees can be laid out in 2D with no wire crossings. This is an attractive property of trees.

Network Topologies: Fat Trees



A fat tree network of 16 processing nodes.

Evaluating
Static Interconnection Networks

- *Diameter:* The distance between the farthest two nodes in the network. The diameter of a linear array is $p - 1$, that of a mesh is $2(\sqrt{p} - 1)$, that of a tree and hypercube is $\log p$, and that of a completely connected network is $O(1)$.

- *Bisection Width:* The minimum number of wires you must cut to divide the network into two equal parts. The bisection width of a linear array and tree is *1*, that of a mesh is $\sqrt{p}$, that of a hypercube is $p/2$ and that of a completely connected network is $p^2/4$.

- *Cost:* The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.

Evaluating Static Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound *k*-ary *d*-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Evaluating Dynamic Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2\log p$	1	2	$p-1$

Communication Costs in Parallel Machines

- Along with idling and contention, communication is a major overhead in parallel programs.

- The cost of communication is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols.

Message Passing Costs in Parallel Computers

- The total time to transfer a message over a network comprises of the following:
  - *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  - *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
  - *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

Store-and-Forward Routing

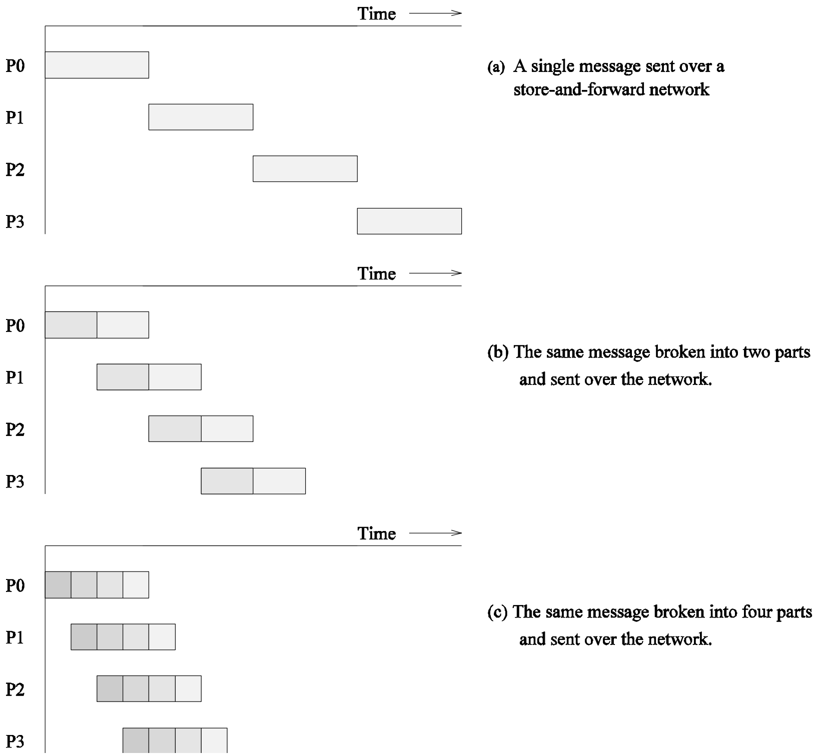
- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size *m* words to traverse *l* communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, t_h is small and the above expression can be approximated by

$$t_{comm} = t_s + mlt_w.$$

+ Routing Techniques


Time
P0
P1
P2
P3
(a) A single message sent over a store-and-forward network
Time
P0
P1
P2
P3
(b) The same message broken into two parts and sent over the network.
Time
P0
P1
P2
P3
(c) The same message broken into four parts and sent over the network.

Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

Packet Routing

- Store-and-forward makes poor use of communication resources.
- Packet routing breaks messages into packets and pipelines them through the network.
- Since packets may take different paths, each packet must carry routing information, error checking, sequencing, and other related header information.
- The total communication time for packet routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

- The factor t_w accounts for overheads in packet headers.

Cut-Through Routing

- Takes the concept of packet routing to an extreme by further dividing messages into basic units called flits.
- Since flits are typically small, the header information must be minimized.
- This is done by forcing all flits to take the same path, in sequence.
- A tracer message first programs all intermediate routers. All flits then take the same route.
- Error checks are performed on the entire message, as opposed to flits.
- No sequence numbers are needed.

Cut-Through Routing

- The total communication time for cut-through routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

- This is identical to packet routing, however, t_w is typically much smaller.

Simplified Cost Model for Communicating Messages

- The cost of communicating a message between two nodes *l* hops away using cut-through routing is given by

$$t_{comm} = t_s + lt_h + t_w m.$$

- In this expression, t_h is typically smaller than t_s and t_w. For this reason, the second term in the RHS does not show, particularly, when *m* is large.

- Furthermore, it is often not possible to control routing and placement of tasks.

- For these reasons, we can approximate the cost of message transfer by

$$t_{comm} = t_s + t_w m.$$

Simplified Cost Model for Communicating Messages

- It is important to note that the original expression for communication time is valid for only uncongested networks.
- If a link takes multiple messages, the corresponding t_w term must be scaled up by the number of messages.
- Different communication patterns congest different networks to varying extents.
- It is important to understand and account for this in the communication time accordingly.

Cost Models for Shared Address Space Machines

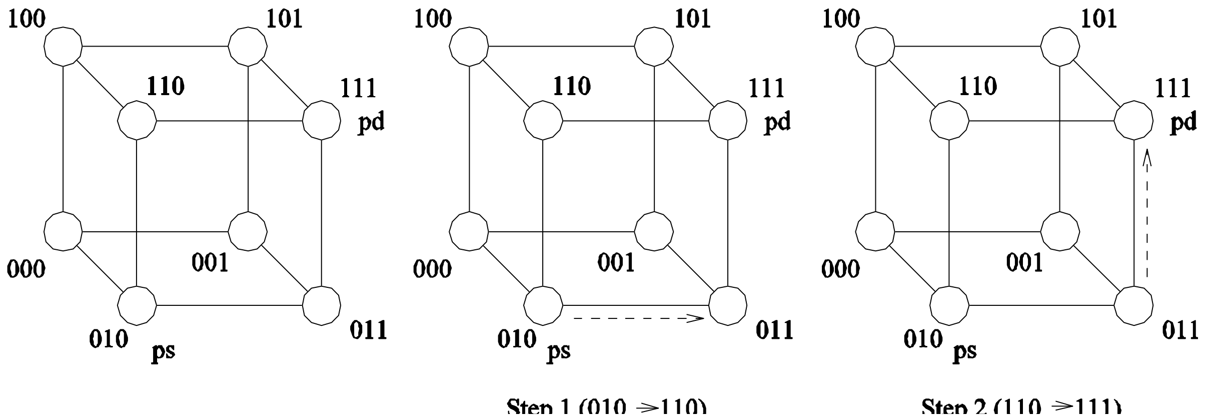
- While the basic messaging cost applies to these machines as well, a number of other factors make accurate cost modeling more difficult.
- Memory layout is typically determined by the system.
- Finite cache sizes can result in cache thrashing.
- Overheads associated with invalidate and update operations are difficult to quantify.
- Spatial locality is difficult to model.
- Prefetching can play a role in reducing the overhead associated with data access.
- False sharing and contention are difficult to model.

Routing Mechanisms for Interconnection Networks

- How does one compute the route that a message takes from source to destination?
  - Routing must prevent deadlocks - for this reason, we use dimension-ordered or e-cube routing.
  - Routing must avoid hot-spots - for this reason, two-step routing is often used. In this case, a message from source *s* to destination *d* is first sent to a randomly chosen intermediate processor *i* and then forwarded to destination *d*.

+

Routing Mechanisms for Interconnection Networks



Routing a message from node P_s (010) to node P_d (111) in a three-dimensional hypercube using E-cube routing.