CZ4102 – High Performance Computing

Lectures 2 and 3: The Hardware Considerations

- Dr Tay Seng Chuan

Reference: ``Introduction to Parallel Computing'' – Chapter 2.

1

Topic Overview

- Implicit Parallelism: Trends in Microprocessor Architectures
- Limitations of Memory System Performance
- Dichotomy of Parallel Computing Platforms
- Communication Model of Parallel Platforms
- Physical Organization of Parallel Platforms
- Communication Costs in Parallel Machines
- Messaging Cost Models and Routing Mechanisms
- Mapping Techniques

2

Scope of Parallelism

- Different applications utilize different aspects of parallelism - e.g., data intensive applications utilize high aggregate throughput, server applications utilize high aggregate network bandwidth, and scientific applications typically utilize high processing and memory system performance.
- It is important to understand each of these performance bottlenecks and their interacting effect.

3

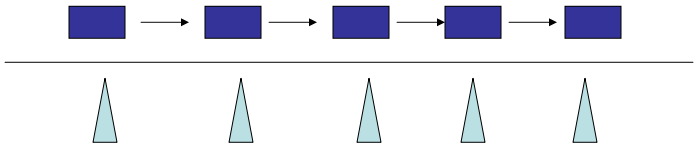
Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
- Higher levels of device integration have made a large number of transistors available.
- The question of how best to utilize these resources is an important one.
- Current processors use these resources in multiple functional units Eg: Add R1, R2 ➔ (i) Instruction Fetch, (ii) Instruction Decode, (iii) Instruction Execute, and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures with different performance and for different purpose. (Each architecture has its own merits and pitfalls.)

4

Pipelining and Superscalar Execution

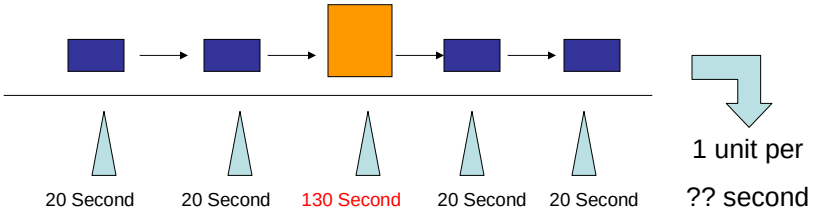
- Pipelining overlaps various stages of instruction execution to achieve performance.
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- This is akin to an assembly line for manufacture of cars.



5

Pipelining and Superscalar Execution

- Pipelining, however, has several limitations.
- The speed of a pipeline is eventually limited by the slowest stage.

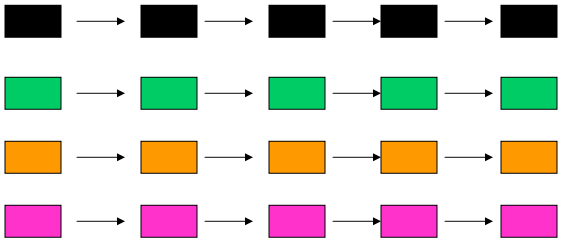


- For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in state-of-the-art Pentium processors).
- However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.
- The penalty of a mis-prediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

6

Pipelining and Superscalar Execution

- One simple way of alleviating these bottlenecks is to use multiple pipelines.



- Selecting which pipeline for execution becomes a challenge.

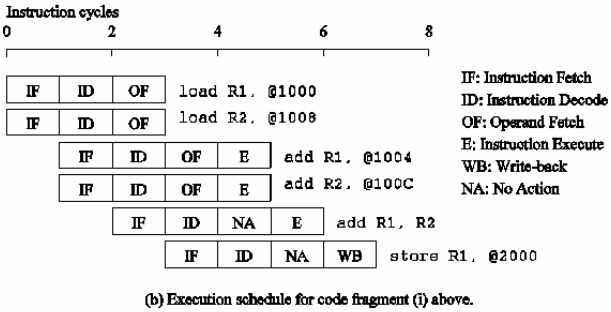
7

Superscalar Execution: An Example

```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.

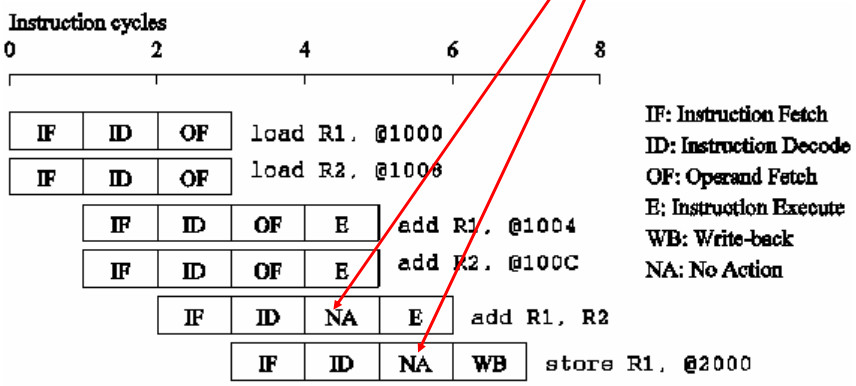


(b) Execution schedule for code fragment (i) above.

Example of a two-way superscalar execution of instructions. 8

Superscalar Execution: An Example

- In the above example, there is some wastage of resources due to data dependencies.



(b) Execution schedule for code fragment (i) above.

Superscalar Execution: An Example

- The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.

```
1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

(i)

```
1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000
```

(ii)

```
1. load R1, @1000
2. add R1, @1004
3. load R2, @1008
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

(iii)

Superscalar Execution

- Scheduling of instructions is determined by a number of factors:
  - **True Data Dependency**: The result of one operation is an input to the next.
  - **Resource Dependency**: Two operations require the same resource.
  - **Branch Dependency**: Scheduling instructions across conditional branch statements cannot be done deterministically before hands.
  - The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
  - The complexity of this hardware is an important constraint on superscalar processors.

11

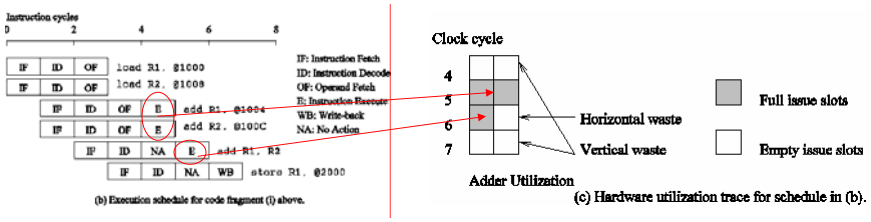
Superscalar Execution: Issue Mechanisms

- In the simpler model, instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called *in-order* issue. (sequentialization)
- In a more aggressive model, instructions can be issued out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue. (speculation)
- Performance of in-order issue is generally limited. Why?

12

Superscalar Execution: Efficiency Considerations

- Not all functional units can be kept busy at all times.
- If during a cycle, no functional units are utilized, this is referred to as vertical waste.



- If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.
- Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.
- Conventional microprocessors typically support four-way superscalar execution.

13

Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
- These instructions are packed and dispatched together, and thus the name very long instruction word.

14

Very Long Instruction Word (VLIW) Processors: Considerations

- Hardware aspect is simpler.
- Compiler has a bigger context from which to select co-scheduled instructions. (More work for compiler.)
- Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.
- Branch and memory prediction is more difficult.
- VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
- Typical VLIW processors are limited to 4-way to 8-way parallelism.

15

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. (Waiting time until the first data is received. 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. If you want immediate response from the hydrant, it is important to reduce latency.)
- Bandwidth is the rate at which data can be pumped to the processor by the memory system. (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 to fight big fires, you need high bandwidth.)

16

Memory Latency: An Example

- Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The following observations follow:
  - Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data. This is a serious drawback.
  - The peak processor rating (assume no memory access) is computed as follows:

    Assume 4 instructions are executed on registers, peak processing rating **= (4 Instructions)/(1 ns) = 4 GFLOPS**. But this is not possible if memory access is needed.

GFLOPS: Giga Floating Point Operations per Second

17

Seriousness of Memory Latency

- On the above architecture with memory latency of 100 ns, consider the problem of computing a dot-product of two vectors.
  - A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch.
  - It follows that the peak speed of this computation is limited to one floating point operation every 100 ns, ie,

    (1 FLOP)/(100 ns) = 1/(100 x 10^{-9} sec) FLOPS = 10^7 FLOPS

    = 10 x 10^6 FLOPS = 10 MFLOPS.
  - The speed of 10 MFLOPS is a very small fraction of the peak processor rating (4 GFLOPS)!

18

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. Eg: For 100 attempts to access to data in cache, if 30 attempts are successful, what is the cache hit ratio?

  What is the cache miss ratio? What is the impact if the cache miss ratio is greater than the cache hit ratio?

19

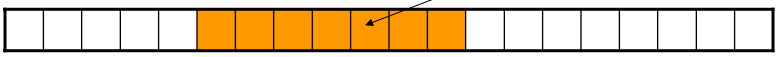
Impact of Memory Bandwidth

- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
- Memory bandwidth can be improved by increasing the size of memory blocks.
- It is important to note that increasing block size does not change latency of the system.
- In practice, wide data and address buses are expensive to construct.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved. The reduces latency by half.

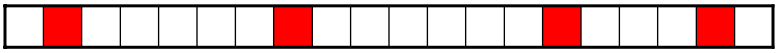
20

Impact of Memory Bandwidth

- Increased bandwidth results can improve computation rates.
- The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions (spatial locality of reference).



Consecutive Data



Scattered Data (strided access)

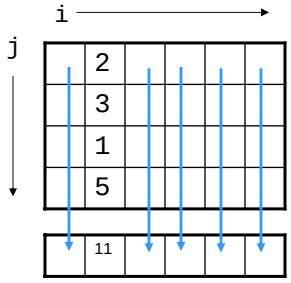
- If we take a data-layout centric view, computations must be reordered to enhance spatial locality of reference. See next 2 examples.

21

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];
```



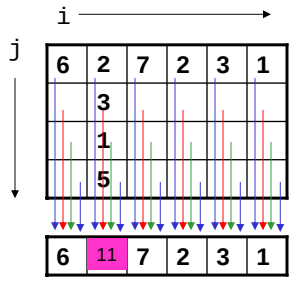
The code fragment sums columns of the matrix b into a vector **column_sum.**

22

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.

23

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 (eg, the two columnsum examples) appropriately can make a significant impact on the spatial and temporal locality.

24

Alternate Approaches for Hiding Memory Latency

- Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:
  - we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
  - we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
  - we access a whole bunch of pages in one go - amortizing the latency across various accesses.
- The first approach is called *prefetching*, the second *multithreading*, and the third one corresponds to *spatial locality* in accessing memory words.

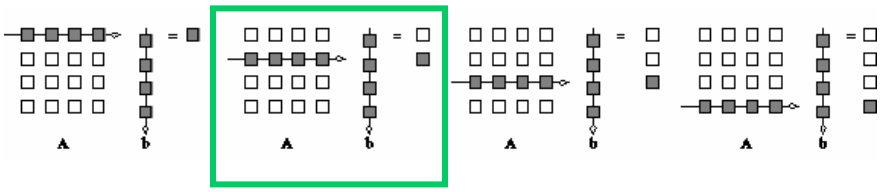
25

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program.

We illustrate threads with a simple example:

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



Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

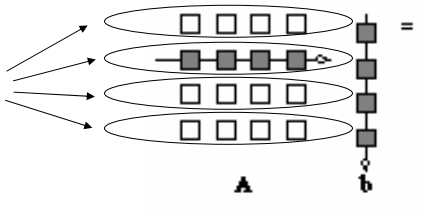
```
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product,get_row(a, i), b);
```

26

Multithreading for Latency Hiding: Example

- In the code, the first instance of this function accesses a pair of vector elements and waits for them.
- In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.

Multiple threads are created.



- After ℓ units of time, where ℓ is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.
- In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation. This is how the memory latency is reduced.

27

Multithreading for Latency Hiding

- The execution schedule in the previous example is predicated upon two assumptions: the memory system is capable of servicing multiple outstanding requests, and the processor is capable of switching threads at every cycle.
- It also requires the program to have an explicit specification of concurrency in the form of threads.

28

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

29

Tradeoffs of Multithreading and Prefetching

- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
- Multithreaded systems become bandwidth bound instead of latency bound. Why?
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem (from bad to worst).
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.
- Context switching overhead incurred by the threads is significant (but is not discussed in textbook).

30

Control Structure of Parallel Programs

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

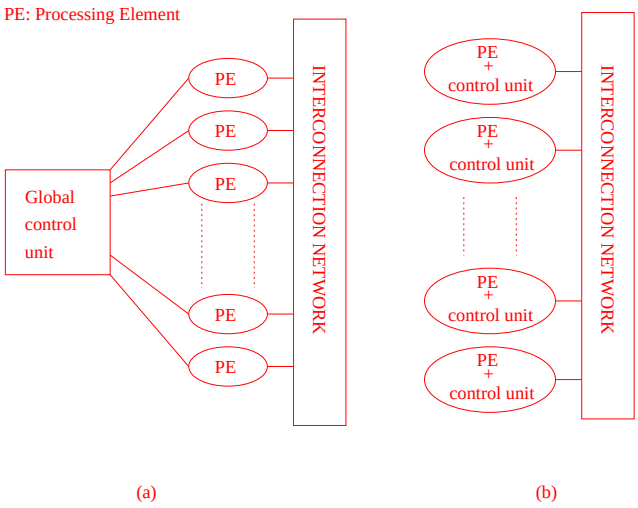
31

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 unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).

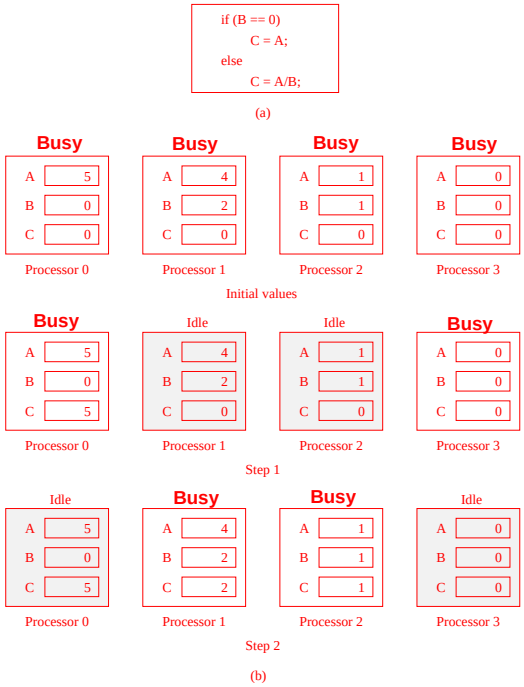
32

SIMD and MIMD Processors



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

33



Conditional Execution in SIMD Processors

Executing a conditional statement on an SIMD computer with four processors:

(a) the conditional statement;

(b) the execution of the statement in two steps.

34

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.

35

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.

36

Shared-Address-Space vs. Shared Memory Machines

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

37

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 (used by CZ4102) and PVM provide such primitives.

38

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

39

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? It depends on the semantic:
  - 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.

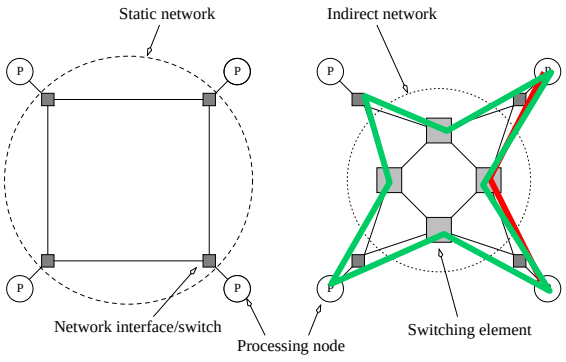
40

Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.
- Interconnectors are made of switches and links (wires, fiber).
- Interconnectors 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. Its configuration cannot be changed.
- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks. Its configuration can be changed.

41

Static and Dynamic Interconnection Networks

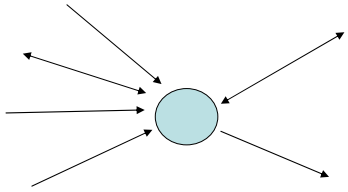


Classification of interconnection networks:
(a) a static network; and (b) a dynamic network.

42

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.



Degree of the switch = ?

43

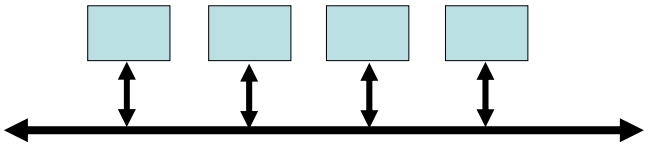
Network Topologies

- 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.

44

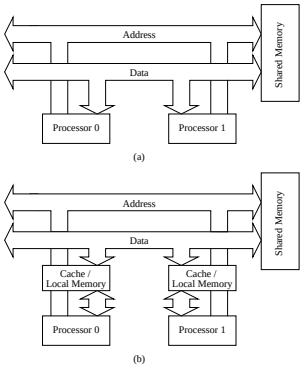
Network Topologies: Buses

- 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.



45

Network Topologies: Buses



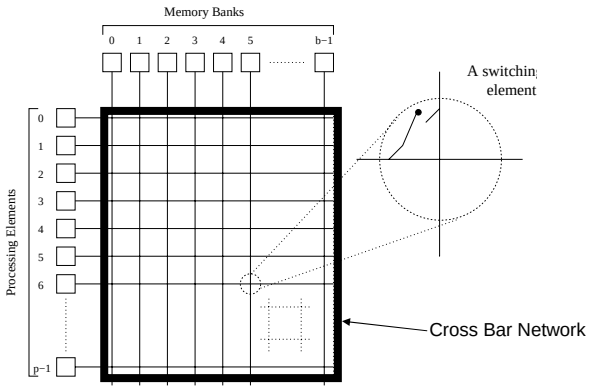
Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

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

46

Network Topologies: Crossbars

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

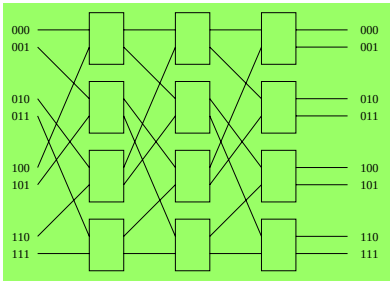


A completely non-blocking crossbar network connecting p processors to b memory banks.

47

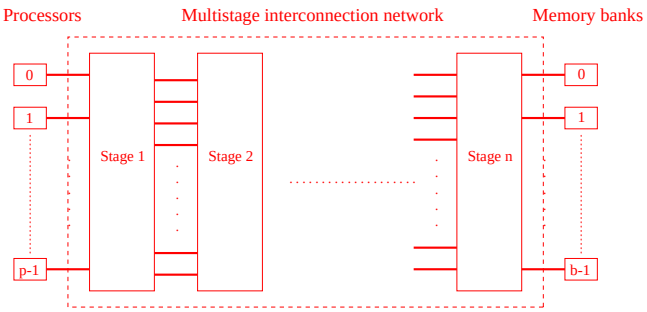
Network Topologies: Multistage Networks

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



48

Network Topologies: Multistage Networks



The schematic of a typical multistage interconnection network.

Network Topologies: Multistage Omega Network

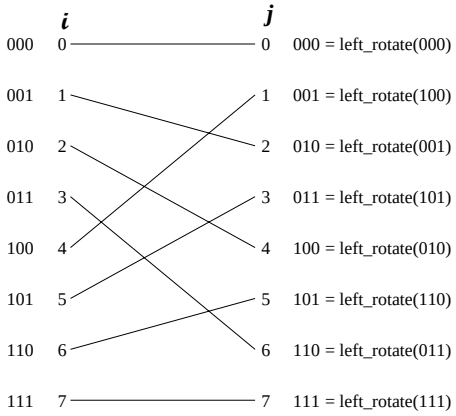
- One of the most commonly used multistage interconnects is the Omega network.
- This network consists of *log p* stages, where *p* is the number of inputs/outputs.
- At each stage, input *i* is connected to output *j* if:

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

Network Topologies: Multistage Omega Network

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

$$j = \begin{cases} 2i, & 0 \le i \le p/2 - 1 \\ 2i \mid 1 \;\; p, & p/2 \le i \le p \;\; 1 \end{cases}$$



A perfect shuffle interconnection for eight inputs and outputs.

51

Network Topologies: Multistage Omega Network

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

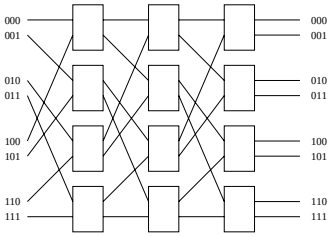
(a) (b)

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

52

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 $p/2 \times \log p$ switching nodes, and the cost of such a network grows as $(p \log p)$.

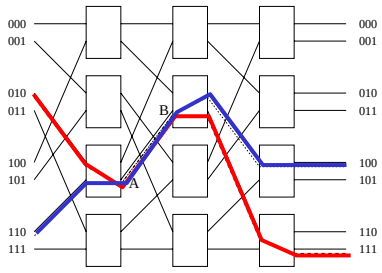
53

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.

54

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.

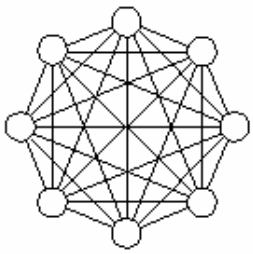
(010 to 111): cross, straight, cross

(110 to 100): straight, cross, straight

55

Network Topologies: Completely Connected Network

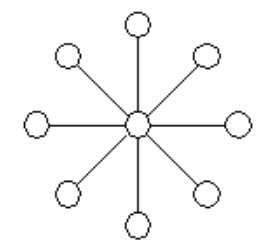
- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of p.
- In this sense, these networks are static counterparts of crossbars.



56

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.

57

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

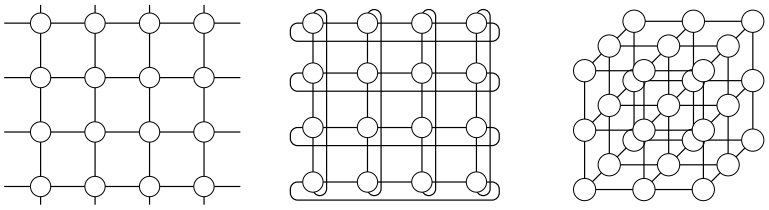
- In a linear array, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a 1-D torus or a ring.

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

58

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

- 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.

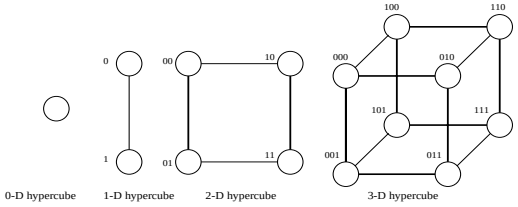


Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

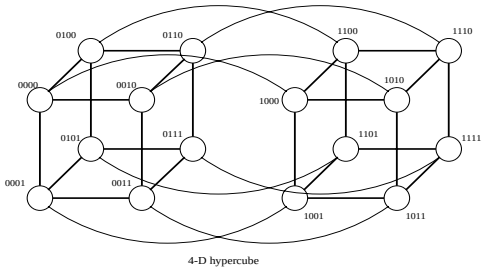
59

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

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



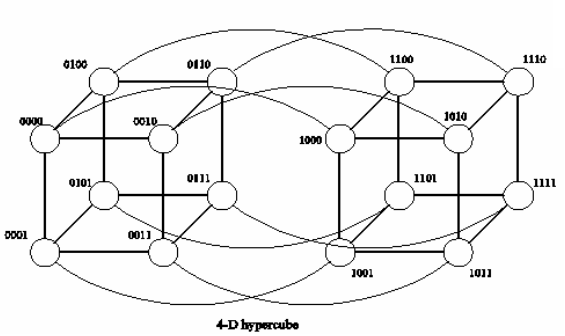
Construction of hypercubes from hypercubes of lower dimension.



60

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.

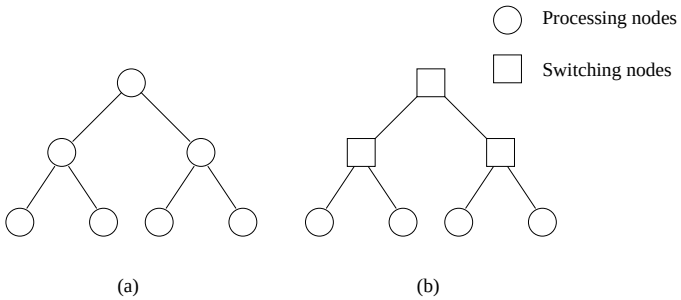


Distance from 0000 to 1000 =1

Distance from 0100 to 1011 = 4

61

Network Topologies: Tree-Based Networks

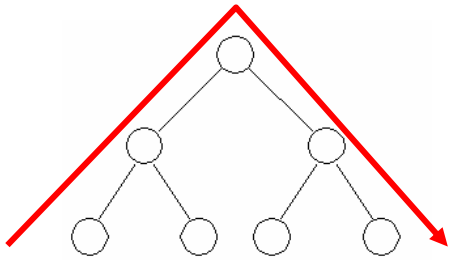


Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

62

Network Topologies: Tree Properties

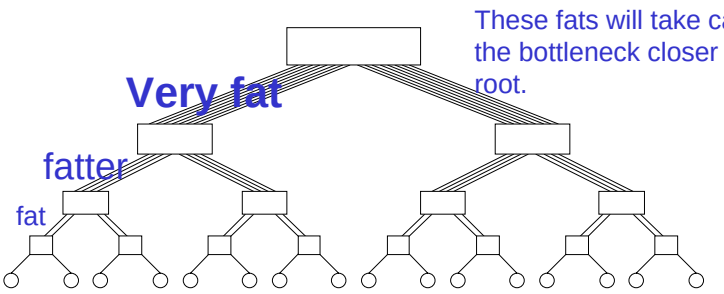
- The distance between any two nodes is no more than *2logp*.



- 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.

63

Network Topologies: Fat Trees

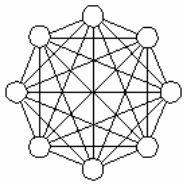
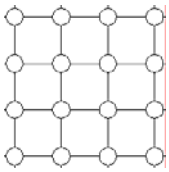
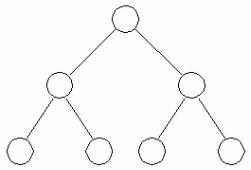
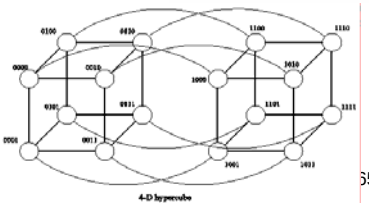


A fat tree network of 16 processing nodes.

64

Evaluating Static Interconnection Networks

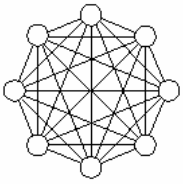
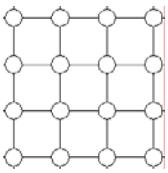
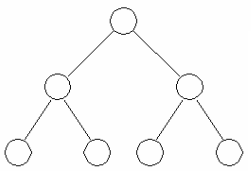
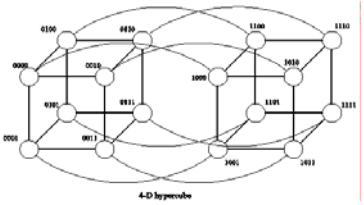
- *Diameter:* The distance between the farthest two nodes in the network. The diameter of a linear array is $p - 1$, that of a mesh is $2(\sqrt{p} - 1)$, that of a tree is O(*log p) (worst case is p),* and hypercube is *log p*, and that of a completely connected network is *O(1).*

55

Evaluating Static Interconnection Networks

- *Bisection Width (may not have a picture to represent):* 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 $p^2/4$.

Evaluating Static Interconnection Networks

- *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.

67

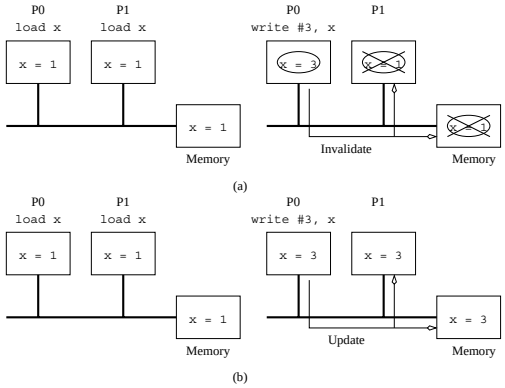
Cache Coherence in Multiprocessor Systems

- Interconnection networks provide basic mechanisms for data transfer.
- The underlying technique must provide some guarantees on the semantics – data integrity must be ensured.
- This guarantee is generally one of serializability, i.e., there exists some serial order of instruction execution that corresponds to the parallel schedule.

68

Cache Coherence in Multiprocessor Systems

When the value of a variable is changes, all its copies must either be invalidated or updated.



Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

69

Cache Coherence: Update and Invalidate Protocols

- If a processor just reads a value once and does not need it again, an update protocol may generate significant overhead. It will be a waste of effort.
- If two processors make interleaved test and updates to a variable, an update protocol is better because the new value will be accessed again.
- Both protocols suffer from false sharing overheads (two words that are not shared, however, they lie on the same cache line).
- Most current machines use invalidate protocols.

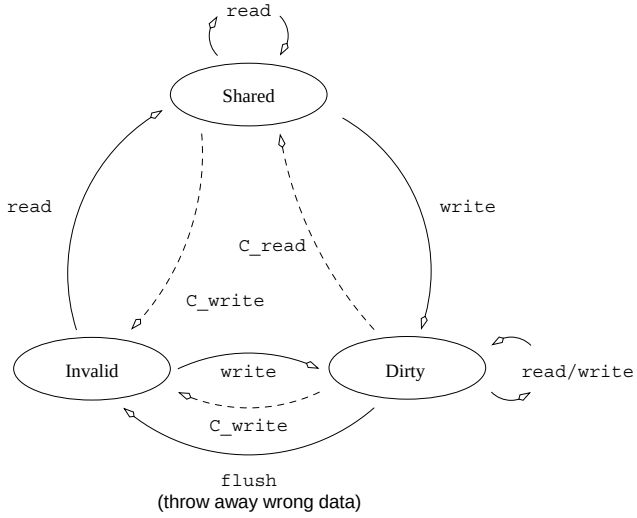
70

Maintaining Coherence Using Invalidate Protocols

- Each copy of a data item is associated with a state.
- One example of such a set of states is, shared, invalid, or dirty.
- In shared state, there are multiple valid copies of the data item (and therefore, an invalidate would have to be generated on an update).
- In dirty state, only one copy exists and therefore, no invalidates need to be generated.
- In invalid state, the data copy is invalid, therefore, a read generates a data request (and associated state changes).

71

Maintaining Coherence Using Invalidate Protocols



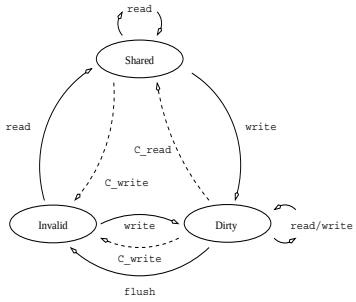
State diagram of a simple three-state coherence protocol.

72

Maintaining Coherence Using Invalidate Protocols

Time ↓

Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x		x = 5, S		x = 5, S y = 12, S
	read y		y = 12, S	
x = x + 1		x = 6, D		x = 5, I y = 12, I
	y = y + 1		y = 13, D	
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I



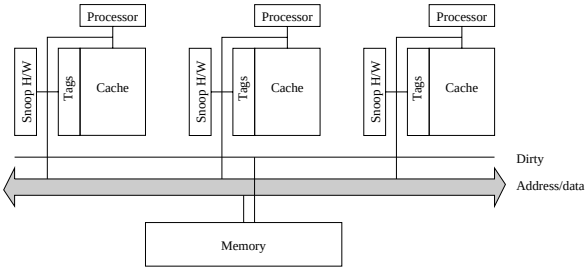
Example of parallel program execution with the simple three-state coherence protocol.

73

Snoopy Cache Systems

How are invalidates sent to the right processors?

In snoopy caches, there is a broadcast media that listens to all invalidates and read requests and performs appropriate coherence operations locally.



A simple snoopy bus based cache coherence system.

74

Performance of Snoopy Caches

- Once copies of data are tagged dirty (have been altered), all subsequent operations can be performed locally (use the latest values) on the cache without generating external traffic.
- If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local (data has been acquired).
- If processors read and update data at the same time, they generate coherence requests on the bus (to update the other copy) - which is ultimately bandwidth limited.

75

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.

76

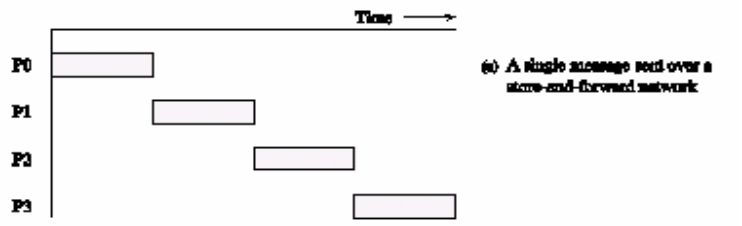
Message Passing Costs in Parallel Computers

- 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 to set up communication link (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.

77

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

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

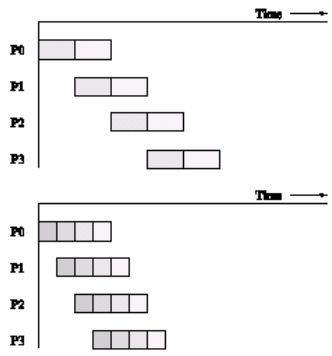
78

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.



79

Cut-Through Routing (Same path for all Flits)

- 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.

80

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

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

81

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.

82

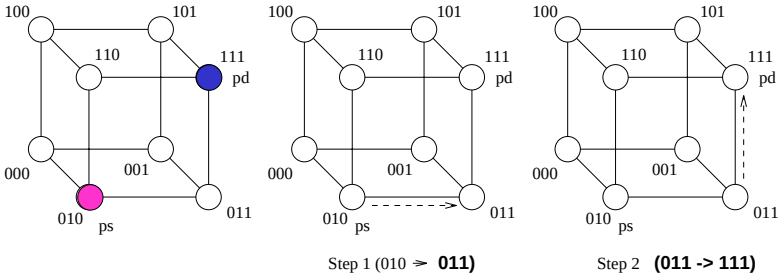
Cost Models for Shared Address Space Machines

- 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.

83

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

84

Mapping Techniques for Graphs

- Often, we need to embed a known communication pattern into a given interconnection topology.
- We may have an algorithm designed for one network, which we are porting to another topology.

For these reasons, it is useful to understand mapping between graphs.

Mapping Techniques for Graphs: Metrics

- When mapping a graph *G(V,E)* into *G'(V',E'),* the following metrics are important:
- The maximum number of edges mapped onto any edge in *E'* is called the *congestion* of the mapping.
- The maximum number of links in *E'* that any edge in *E* is mapped onto is called the *dilation* of the mapping.
- The ratio of the number of nodes in the set *V'* to that in set *V* is called the *expansion* of the mapping.

Embedding a Linear Array into a Hypercube

- A linear array (or a ring) composed of 2^d nodes (labeled 0 through $2^d - 1$) can be embedded into a *d*-dimensional hypercube by mapping node *i* of the linear array onto node
- *G(i, d)* of the hypercube. The function *G(i, x)* is defined as follows:

$$
\begin{aligned}
G(0,1) &= 0 \\
G(1,1) &= 1 \\
G(i,x+1) &= \begin{cases} G(i,x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}
\end{aligned}
$$

87

Embedding a Linear Array into a Hypercube

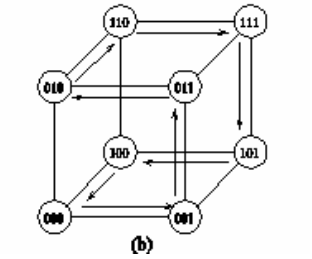
The function *G* is called the *binary reflected Gray code* (RGC).

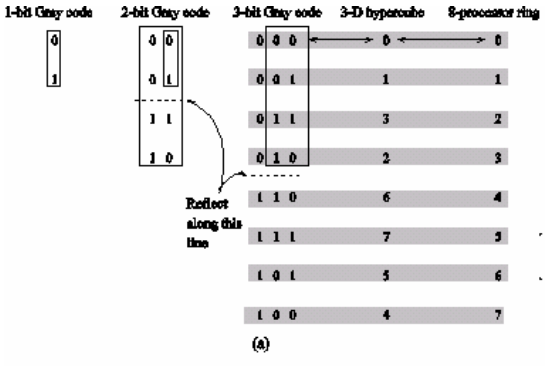
Since adjoining entries (*G(i, d)* and *G(i + 1, d)*) differ from each other at only one bit position, corresponding processors are mapped to neighbors in a hypercube. Therefore, the congestion, dilation, and expansion of the mapping are all 1.

$$
\begin{aligned}
G(0,1) &= 0 \\
G(1,1) &= 1 \\
G(i,x+1) &= \begin{cases} G(i,x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}
\end{aligned}
$$

88

Embedding a Linear Array into a Hypercube: Example





$$G(0,1) = 0$$
$$G(1,1) = 1$$
$$G(i,x+1) = \begin{cases} G(i,x), & i < 2^x \\ 2^x + G(2^{x+1}-1-i,x), & i \geq 2^x \end{cases}$$

(a) A three-bit reflected Gray code ring; and (b) its embedding into a three-dimensional hypercube.

89