Chapter 2: Parallel Programming Platforms

Introduction to Parallel Computing, Second Edition
By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar
Contents

- Implicit Parallelism: Trends in Microprocessor Architectures
- Limitations of Memory System Performance
- Dichotomy of Parallel Computing Platforms
- Physical Organization of Parallel Platforms
- Communication Costs in Parallel Machines
- Routing Mechanisms for Interconnection Networks
- Impact of Process-Processor Mapping and Mapping Techniques
A serial and implicitly parallel architectures and an overview of important architectural concepts as they relate to parallel processing.
Parallel programming platforms Provides sufficient detail for programmers to be able to write efficient code on a variety of platforms.
By developing cost models and abstractions for quantifying the performance of various parallel algorithms, and identify bottlenecks resulting from various programming constructs.

2.1 Implicit Parallelism: Trends in Microprocessor Architectures

Clock speeds of microprocessors has improved -two to three orders of magnitude over the past 20 years. These increments in clock speed don’t fulfill memory technology limitations.
Transistors are consequences of the integration techniques that enable execution of multiple instructions in a single clock cycle.
The mechanisms used by various processors for supporting multiple instruction execution are discussed in the following sections:
2.1.1 Pipelining and Superscalar Execution:

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store, among others), pipelining enables faster execution.

The speed of a single pipeline is limited by the largest atomic task in the pipeline. In typical instruction traces, every fifth to sixth instruction is a branch instruction. Long instruction pipelines need effective techniques for predicting branch destinations so that pipelines can be speculatively filled.

The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed. These factors place limitations on the depth of a processor pipeline and the resulting performance gains.

A way to improve the performance beyond this level is to use multiple pipelines. An example to illustrate pipelining:
Example 2.1 Superscalar execution:
The ability of a processor to issue multiple instructions in the same cycle is referred to as superscalar execution. Consider a processor with two pipelines and the ability to simultaneously issue two instructions.
Figure 2.1. Example of a two-way superscalar execution of instructions.

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

<table>
<thead>
<tr>
<th>Instruction cycles</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF: Instruction Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID: Instruction Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF: Operand Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E: Instruction Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB: Write−back</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NA: No Action</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

```
Clock cycle
4
5
6
7
```

Adder Utilization

(c) Hardware utilization trace for schedule in (b).
2.1.2 Very Long Instruction Word Processors

The parallelism extracted by superscalar processors is often limited by the instruction look-ahead. Alternate concept for exploiting instruction-level parallelism used in very long instruction word (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time.

Instructions that can be executed concurrently are packed into groups and parceled off to the processor as a single long instruction word to be executed on multiple functional units at the same time.

Scheduling is done in software, the decoding and instruction issue mechanisms are simpler in VLIW processors.

The compiler has a larger context from which to select instructions and can use a variety of transformations to optimize parallelism when compared to a hardware issue unit.

The performance of VLIW processors is very sensitive to the compilers ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism.
Loop unrolling, branch prediction and speculative execution all play important roles in the performance of VLIW processors. While superscalar and VLIW processors have been successful in exploiting implicit parallelism, they are generally limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

2.2 Limitations of Memory System Performance

The effective performance of a program on a computer relies not just on the speed of the processor but also on the ability of the memory system to feed data to the processor. At the logical level, a memory system, possibly consisting of multiple levels of caches, takes in a request for a memory word and returns a block of data of size $b$ containing the requested word after $l$ nanoseconds.

Here, $l$ is referred to as the latency (latency is the delay from input into a system to desired outcome) of the memory. Bandwidth of the memory system is the rate at which data can be pumped from the memory to the processor.
2.2.1 Improving Effective Memory Latency Using Caches

Cache is placed between the processor and DRAM, which acts as a low-latency high-bandwidth storage. The data needed by the processor is first fetched into the cache. All subsequent accesses to data items residing in the cache are serviced by the cache.

In principle, 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. The performance of memory bound programs is critically impacted by the cache hit ratio.

The effective computation rate of many applications is bounded not by the processing rate of the CPU but by the rate at which data can be pumped into the CPU known as **memory bound**.
The repeated reference to a data item in a small time window is called temporal locality of the reference which is main assumption for the improvement in performance by the presence of cache.

2.2.2 Impact of Memory Bandwidth

Memory bandwidth refers to the rate at which data can be moved between the processor and memory. It is determined by the bandwidth of the memory bus as well as the memory units.

One commonly used technique to improve memory bandwidth is to increase the size of the memory blocks. Assume that a single memory request returns a contiguous block of four words. The single unit of four words in this case is referred to as a cache line.
Example 2.5 Impact of strided access
Consider the following code fragment:

```c
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. There are two observations can be made: (i) the vector column_sum is small and easily fits into the cache; and (ii) the matrix b is accessed in a column order as illustrated in Figure 2.2(a).
Figure 2.2. 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.
2.2.3 Alternate Approaches for Hiding Memory Latency

Browsing the web during peak network traffic hours, the lack of response from your browser can be alleviated using one of three simple approaches:

(i) anticipate which pages are going to browse ahead of time and issue requests for them in advance; can be called as prefetching.

(ii) we open multiple browsers and access different pages in each browser, while waiting for one page to load, called as multithreading.

(iii) Access a whole bunch of pages in one go – amortizing the latency across various accesses, called as spatial locality in accessing memory words.

Multithreading for Latency Hiding

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

Example 2.7 Threaded execution of matrix multiplication

Consider the following code segment for multiplying an $n \times n$ matrix $a$ by a vector $b$ to get vector $c$.

1 $\text{for}(i=0;i<n;i++)$
2 $c[i] = \text{dot_product(get_row(a, i), b);}$
This code computes each element of c as the dot product of the corresponding row of a with the vector b. Notice that each dot-product is independent of the other, and therefore represents a concurrent unit of execution.

We can rewrite the above code segment as:

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

The only difference between the two code segments is that we have explicitly specified each instance of the dot-product computation as being a thread. Now, consider the execution of each instance of the function dot_product. 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. 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. In this way, computation can be performed in every clock cycle.
Prefetching for Latency Hiding
In a program, a data item is loaded and used by a processor in a small time window. If the load results in a cache miss, then the use stalls. A simple solution to this problem is to advance the load operation so that even if there is a cache miss, the data is likely to have arrived by the time it is used. However, if the data item has been overwritten between load and use, a fresh load is issued.

Example 2.8 Hiding latency by prefetching
Consider the problem of adding two vectors $a$ and $b$ using a single for loop. In the first iteration of the loop, the processor requests $a[0]$ and $b[0]$. Since these are not in the cache, the processor must pay the memory latency. While these requests are being serviced, the processor also requests $a[1]$ and $b[1]$. Assuming that each request is generated in one cycle (1 ns) and memory requests are satisfied in 100 ns, after 100 such requests the first set of data items is returned by the memory system. Subsequently, one pair of vector components will be returned every cycle. In this way, in each subsequent cycle, one addition can be performed and processor cycles are not wasted.
2.2.4 Tradeoffs of Multithreading and Prefetching

It might seem that Multithreading and prefetching solve all the problems related to memory system performance, they are critically impacted by the memory bandwidth.

2.3 Dichotomy of Parallel Computing Platforms

A dichotomy is based on the logical and physical organization of parallel platforms. The logical organization refers to a programmer’s view of the platform while the physical organization refers to the actual hardware organization of the platform. The two critical components of parallel computing from a programmer's perspective are ways of expressing parallel tasks (control structure) and mechanisms for specifying interaction between these tasks (communication model).
2.3.1 Control Structure of Parallel Platforms

Parallel tasks can be specified at various levels of granularity.

i) Each program in a set of programs can be viewed as one parallel task.

ii) Individual instructions within a program can be viewed as parallel tasks.

Between these (i) and (ii) lie a range of models for specifying the control structure of programs and the corresponding architectural support for them. Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.

In architectures referred to as single instruction stream, multiple data stream (SIMD), a single control unit dispatches instructions to each processing unit. Figure 2.3(a) illustrates a typical SIMD architecture.

In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. Computers in which each processing element is capable of executing a different program independent of the other processing elements are called multiple instruction stream, multiple data stream (MIMD) computers.

SIMD computers require less memory because only one copy of the program needs to be stored. MIMD computers store the program and operating system at each processor.
Figure 2.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).

PE: Processing Element
Example 2.11 Execution of conditional statements on a SIMD architecture

Consider the execution of a conditional statement illustrated in Figure 2.4. The conditional statement in Figure 2.4(a) is executed in two steps. In the first step, all processors that have B equal to zero execute the instruction \( C = A \). All other processors are idle. In the second step, the 'else' part of the instruction \( C = A/B \) is executed.

Figure 2.4. Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

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

(a)
2.3.2 Communication Model of Parallel Platforms

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

Shared-Address-Space Platforms

The "shared-address-space" view of a parallel platform supports a common data space that is accessible to all processors. Shared-address-space platforms supporting SPMD programming are also referred to as multiprocessors.

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) multicomputer.

On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a non-uniform memory access (NUMA) multicomputer.
Figure 2.5. 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.
Message-Passing Platforms

The logical machine view of a message-passing platform consists of $p$ processing nodes, each with its own exclusive address space.

Each of these processing nodes can either be single processors or a shared-address-space multiprocessor – a trend that is fast gaining momentum in modern message-passing parallel computers.

Instances of such a view come naturally from clustered workstations and non-shared-address-space multi-computers. On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name message passing.

Since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are $send$ and $receive$ (the corresponding calls may differ across APIs but the semantics are largely identical.

A function $whoami$, is used to make id for target address available for message passing. Another function $numprocs$, specifies the number of processes participating in the ensemble.
2.4 Physical Organization of Parallel Platforms

An ideal architecture, practical difficulties associated with realizing this model and conventional architecture.

2.4.1 Architecture of an Ideal Parallel Computer

A natural extension of the serial model of computation (the Random Access Machine, or RAM) consists of $p$ processors and a global memory of unbounded size that is uniformly accessible to all processors.

All processors access the same address space. Processors share a common clock but may execute different instructions in each cycle.

This ideal model is also referred to as a parallel random access machine (PRAM). Since PRAMs allow concurrent access to various memory locations, depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.
1. Exclusive-read, exclusive-write (EREW) PRAM: In this class, access to a memory location is exclusive. No concurrent read or write operations are allowed. This is the weakest PRAM model, affording minimum concurrency in memory access.

2. Concurrent-read, exclusive-write (CREW) PRAM: In this class, multiple read accesses to a memory location are allowed. However, multiple write accesses to a memory location are serialized.

3. Exclusive-read, concurrent-write (ERCW) PRAM. Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized.

4. Concurrent-read, concurrent-write (CRCW) PRAM. This class allows multiple read and write accesses to a common memory location. This is the most powerful PRAM model.
Allowing concurrent read access does not create any semantic discrepancies in the program. Concurrent write access to a memory location requires arbitration.

Several protocols are used to resolve concurrent writes. The most frequently used protocols are as follows:

- **Common**, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- **Arbitrary**, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- **Priority**, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.
- **Sum**, in which the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).
Architectural Complexity of the Ideal Model

Consider the implementation of an EREW PRAM as a shared-memory computer with p processors and a global memory of m words.

The processors are connected to the memory through a set of switches. These switches determine the memory word being accessed by each processor.

In an EREW PRAM, each of the p processors in the ensemble can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously. To ensure such connectivity, the total number of switches must be $\Theta(mp)$.

2.4.2 Interconnection Networks for Parallel Computers

Interconnection networks provide mechanisms for data transfer between processing nodes or between processors and memory modules. A black box view of an interconnection network consists of n inputs and m outputs.

Interconnection networks are built using links and switches. A link corresponds to physical media such as a set of wires or fibers capable of carrying information.
Interconnection networks can be 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 also known as indirect network, are built using switches and communication links. Figure 2.6. Classification of interconnection networks: (a) a static network; and (b) a dynamic network.
A single switch in an interconnection network consists of a set of input ports and a set of output ports.

Switches provide a range of functionality. The minimal functionality provided by a switch is a mapping from the input to the output ports.

The total number of ports on a switch is also called the degree of the switch.

2.4.3 Network Topologies

There are different types of network topologies existing in network and they are discussed below:

- Bus-Based Networks
- Crossbar Networks
- Multistage Networks
- Completely-Connected Network
- Star-Connected Network
Bus-Based Networks

A bus-based network is simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network scales linearly as the number of nodes, p.

Since the transmission medium is shared, there is little overhead associated with broadcast information between nodes, compared to point-to-point message transfer.

Figure 2.7. Bus-based interconnects (a) with no local caches; (b) with local memory/caches.
Crossbar Networks

A crossbar network employs a grid of switches or switching nodes.

The crossbar network is a non-blocking network in the sense that the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.
Figure 2.8. A completely non-blocking crossbar network connecting \( p \) processors to \( b \) memory banks.
Multistage Networks

The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost.

The shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called **multistage** interconnection networks lies between these two extremes.

It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

A commonly used multistage connection network is the omega network.
Figure 2.9. The schematic of a typical multistage interconnection network.
Figure 2.10. A perfect shuffle interconnection for eight inputs and outputs.
Figure 2.11. Two switching configurations of the 2 x 2 switch: (a) Pass-through; (b) Cross-over.

(a)  
(b)

Figure 2.12. A complete omega network connecting eight inputs and eight outputs.
Figure 2.13. An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.
Figure 2.13 shows data routing over an omega network from processor two (010) to memory bank seven (111) and from processor six (110) to memory bank four (100). This figure also illustrates an important property of this network. When processor two (010) is communicating with memory bank seven (111), it blocks the path from processor six (110) to memory bank four (100). Communication link AB is used by both communication paths. Thus, in an omega network, access to a memory bank by a processor may disallow access to another memory bank by another processor, this is called blocking networks.

**Completely-Connected Network**

In a completely-connected network, each node has a direct communication link to every other node in the network. Completely-connected networks are the static counterparts of crossbar switching networks, since in both networks; the communication between any input/output pair does not block communication between any other pair.

Figure 2.14. (a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.
Star-Connected Network

In a star-connected network, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor.

Communication between any pair of processors is routed through the central processor, just as the shared bus forms the medium for all communication in a bus-based network. The central processor is the bottleneck in the star topology.

Linear Arrays, Meshes, and k-d Meshes

Due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers.

A family of such networks spans the space of linear arrays and hyper cubes.
A linear array is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right.

Figure 2.15. Linear arrays: (a) with no wraparound links; (b) with wraparound link.

A two-dimensional mesh illustrated in Figure 2.16(a) is an extension of the linear array to two-dimensions. Each dimension has nodes with a node identified by a two-tuple \((i, j)\).

A variety of regularly structured computations map very naturally to a 2-D mesh. For this reason, 2-D meshes were often used as interconnects in parallel machines. The three-dimensional cube is a generalization of the 2-D mesh to three dimensions.

A variety of physical simulations commonly executed on parallel computers (for example, 3-D weather modeling, structural modeling, etc.) can be mapped naturally to 3-D network topologies. For this reason, 3-D cubes are used commonly in interconnection networks for parallel computers.
Figure 2.16. 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.

A one-dimensional hypercube is constructed from two zero-dimensional hyper cubes by connecting them.

A two-dimensional hypercube of four nodes is constructed from two one dimensional hyper cubes by connecting corresponding nodes.

A d-dimensional hypercube is constructed by connecting corresponding nodes of two (d - 1) dimensional hyper cubes. Figure 2.17 illustrates this for up to 16 nodes in a 4-D hypercube.
Figure 2.17. Construction of hyper cubes from hyper cubes of lower dimension.
Tree-Based Networks

A tree network is one in which there is only one path between any pair of nodes.

Both linear arrays and star-connected networks are special cases of tree networks. Static tree networks have a processing element at each node of the tree.

In a dynamic tree network, nodes at intermediate levels are switching nodes and the leaf nodes are processing elements.

Figure 2.18. Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.
Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many nodes in the left sub-tree of a node communicate with nodes in the right sub-tree, the root node must handle all the messages.

This problem can be alleviated in dynamic tree networks by increasing the number of communication links and switching nodes closer to the root. This network, also called a fat tree.

Figure 2.19. A fat tree network of 16 processing nodes.
2.4.4 Evaluating Static Interconnection Networks

Various criteria is used to characterize the cost and performance of static interconnection networks.

Diameter

The diameter of a network is the maximum distance between any two processing nodes in the network.

Connectivity

The connectivity of a network is a measure of the multiplicity of paths between any two processing nodes. One measure of connectivity is the minimum number of arcs that must be removed from the network to break it into two disconnected networks which is called the **arc connectivity** of the network.

Bisection Width and Bisection Bandwidth

The **bisection width** of a network is defined as the minimum number of communication links that must be removed to partition the network into two equal halves.
The number of bits that can be communicated simultaneously over a link connecting two nodes is called the **channel width**.

The peak rate at which a single physical wire can deliver bits is called the **channel rate**. The peak rate at which data can be communicated between the ends of a communication link is called **channel bandwidth**.

Channel bandwidth is the product of channel rate and channel width.

The **bisection bandwidth** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as **crosssection bandwidth**.

**Cost**

Many criteria are used to evaluate the cost of a network. One way of defining the cost of a network is in terms of the number of communication links or the number of wires required by the network.

The bisection bandwidth of a network can also be used as a measure of its cost, as it provides a lower bound on the area in a two-dimensional packaging or the volume in a three-dimensional packaging.
Table 2.1. A summary of the characteristics of various static network topologies connecting \( p \) nodes.

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Arc Connectivity</th>
<th>Cost (No. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely-connected</td>
<td>1</td>
<td>( \frac{p^2}{4} )</td>
<td>( p - 1 )</td>
<td>( \frac{p(p - 1)}{2} )</td>
</tr>
<tr>
<td>Star</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>( p - 1 )</td>
</tr>
<tr>
<td>Complete binary tree</td>
<td>( 2 \log((p + 1)/2) )</td>
<td>1</td>
<td>1</td>
<td>( p - 1 )</td>
</tr>
<tr>
<td>Linear array</td>
<td>( p - 1 )</td>
<td>1</td>
<td>1</td>
<td>( p - 1 )</td>
</tr>
<tr>
<td>2-D mesh, no wraparound</td>
<td>( 2(\sqrt{p} - 1) )</td>
<td>( \sqrt{p} )</td>
<td>2</td>
<td>( 2(p - \sqrt{p}) )</td>
</tr>
<tr>
<td>2-D wraparound mesh</td>
<td>( 2\lfloor\sqrt{p}/2\rfloor )</td>
<td>( 2\sqrt{p} )</td>
<td>4</td>
<td>( 2p )</td>
</tr>
<tr>
<td>Hypercube</td>
<td>( \log p )</td>
<td>( p/2 )</td>
<td>( \log p )</td>
<td>( (p \log p)/2 )</td>
</tr>
<tr>
<td>Wraparound ( k )-ary ( d )-cube</td>
<td>( d\lfloor k/2 \rfloor )</td>
<td>( 2k^{d-1} )</td>
<td>( 2d )</td>
<td>( dp )</td>
</tr>
</tbody>
</table>
### 2.4.5 Evaluating Dynamic Interconnection Networks

A number of evaluation metrics for dynamic networks follow from the corresponding metrics for static networks.

**Diameter**

The maximum distance between any two nodes in the network. This is indicative of the maximum delay that a message will encounter in being communicated between the selected pair of nodes. For all networks of interest, this is equivalent to the maximum distance between any (processing or switching) pair of nodes.

**Connectivity**

In terms of node or edge connectivity, the node connectivity is the minimum number of nodes that must fail (be removed from the network) to fragment the network into two parts. We should consider only switching nodes (as opposed to all nodes).

**Bisection width**

In the case of bisection width, consider any possible partitioning of the $p$ processing nodes into two equal parts. Select an induced partitioning of the switching nodes such that the number of edges crossing this partition is minimized.

The minimum number of edges for any such partition is the bisection width of the dynamic network.

Another way, bisection width in terms of the minimum number of edges that must be removed from the network so as to partition the network into two halves with identical number of processing nodes.
Example 2.13 Bisection width of dynamic networks

Considering Network illustrated in figure 2.20. Let’s take three bisections, A, B, and C, each of which partitions the network into two groups of two processing nodes each. Notice that these partitions need not partition the network nodes equally. In the example, each partition results in an edge cut of four. We conclude that the bisection width of this graph is four.

Figure 2.20. Bisection width of a dynamic network is computed by examining various equi-partitions of the processing nodes and selecting the minimum number of edges crossing the partition. In this case, each partition yields an edge cut of four. Therefore, the bisection width of this graph is four.
2.4.6 Cache Coherence in Multiprocessor Systems

Interconnection networks provide basic mechanisms for communicating messages (data), in
the case of shared-address-space computers additional hardware is required to keep multiple copies of
data consistent with each other.

The problem of keeping caches in multiprocessor systems coherent is significantly more
complex than in uniprocessor systems. This is because in addition to multiple copies as in uniprocessor
systems, there may also be multiple processors modifying these copies. When a processor changes the
value of its copy of the variable, one of two things must happen: the other copies must be invalidated, or
the other copies must be updated. Failing this, other processors may potentially work with incorrect
(stale) values of the variable. These two protocols are referred to as invalidate and update protocols.

In an update protocol, whenever a data item is written, all of its copies in the system are
updated.

On the other hand, an invalidate protocol invalidates the data item on the first update at a
remote processor and subsequent updates need not be performed on this copy.

False sharing refers to the situation in which different processors update different parts of the
same cache-line.
Figure 2.21. Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.
Maintaining Coherence Using Invalidate Protocols

Multiple copies of a single data item are kept consistent by keeping track of the number of copies and the state of each of these copies. Here is one possible set of states associated with data items and events that trigger transitions among these states.

Let us revisit the example in Figure 2.21. Initially the variable $x$ resides in the global memory. The first step executed by both processors is a load operation on this variable. At this point, the state of the variable is said to be **shared**, since it is shared by multiple processors. When processor $P_0$ executes a store on this variable, it marks all other copies of this variable as **invalid**. It must also mark its own copy as modified or dirty. This is done to ensure that all subsequent accesses to this variable at other processors will be serviced by processor $P_0$ and not from the memory.

The complete state diagram of a simple three-state protocol is illustrated in Figure 2.22. The solid lines depict processor actions and the dashed lines coherence actions.
Figure 2.22. State diagram of a simple three-state coherence protocol.
Figure 2.23. Example of parallel program execution with the simple three-state coherence protocol discussed in Section 2.4.6.

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction at Processor 0</th>
<th>Instruction at Processor 1</th>
<th>Variables and their states at Processor 0</th>
<th>Variables and their states at Processor 1</th>
<th>Variables and their states in Global mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read x</td>
<td>read y</td>
<td>x = 5, S</td>
<td>y = 12, S</td>
<td>x = 5, S</td>
</tr>
<tr>
<td></td>
<td>x = x + 1</td>
<td>y = y + 1</td>
<td>x = 6, D</td>
<td>y = 13, D</td>
<td>y = 12, S</td>
</tr>
<tr>
<td></td>
<td>read y</td>
<td></td>
<td>y = 13, S</td>
<td></td>
<td>x = 5, I</td>
</tr>
<tr>
<td></td>
<td>read x</td>
<td></td>
<td>x = 6, S</td>
<td></td>
<td>y = 12, I</td>
</tr>
<tr>
<td></td>
<td>x = x + y</td>
<td></td>
<td>x = 19, D</td>
<td></td>
<td>y = 13, S</td>
</tr>
<tr>
<td></td>
<td>y = x + y</td>
<td></td>
<td>x = 6, I</td>
<td></td>
<td>y = 13, I</td>
</tr>
<tr>
<td></td>
<td>x = x + 1</td>
<td>y = y + 1</td>
<td>x = 20, D</td>
<td></td>
<td>x = 6, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y = 20, D</td>
<td></td>
<td>y = 13, I</td>
</tr>
</tbody>
</table>
Snoopy Cache Systems

Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. In such systems, all processors snoop on (monitor) the bus for transactions. This allows the processor to make state transitions for its cache-blocks. Each processor's cache has a set of tag bits associated with it that determine the state of the cache blocks. These tags are updated according to the state diagram associated with the coherence protocol.

Performance of Snoopy Caches

The performance gains of snoopy systems are derived from the fact that if different processors operate on different data items, these items can be cached.

Snoopy protocols are intimately tied to multicomputer based on broadcast networks such as buses because all processors must snoop all the messages. This is not scalable option so, An obvious solution to this problem is to propagate coherence operations only to those processors that must participate in the operation. This solution requires us to keep track of which processors have copies of various data items and also the relevant state information for these data items. This information is stored in a directory, and the coherence mechanism based on such information is called a directory-based system.
Figure 2.24. A simple snoopy bus based cache coherence system.
Directory Based Systems

Consider a simple system in which the global memory is augmented with a directory that maintains a bitmap (presence bits) representing cache-blocks and the processors at which they are cached. The key to the performance of directory based schemes is the simple observation that only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations.

Performance of Directory Based Schemes

Coherence actions are initiated when multiple processors attempt to update the same data item. In this case, in addition to the necessary data movement, coherence operations add to the overhead in the form of propagation of state updates (invalidates or updates) and generation of state information from the directory.

If a parallel program requires a large number of coherence actions (large number of read/write shared data blocks) the directory will ultimately bound its parallel performance. From the point of view of cost, the amount of memory required to store the directory may itself become a bottleneck as the number of processors increases. One solution would be to make the memory block larger (thus reducing \( m \) for a given memory size). However, this adds to other overheads such as false sharing.
Figure 2.25. Architecture of typical directory based systems: (a) a centralized directory; and (b) a distributed directory.
Distributed Directory Schemes

In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks, its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. Similarly, when a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. In this way, the directory is decentralized and the contention associated with the central directory is alleviated. The communication overhead associated with state update messages is not reduced.

Performance of Distributed Directory Schemes

Distributed directories permit $O(p)$ simultaneous coherence operations, the underlying network can sustain the associated state update messages. Distributed directories are inherently more scalable than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.
2.5 Communication Costs in Parallel Machines

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.

2.5.1 Message Passing Costs in Parallel Computers

The time taken to communicate a message between two nodes in a network is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination.

The principal parameters that determine the communication latency are as follows: Startup time ($ts$): time required to handle a message at the sending and receiving nodes. Per-hop time ($th$): The time taken by the header of a message to travel between two directly-connected nodes in the network. Per-word transfer time ($tw$): If the channel bandwidth is $r$ words per second, then each word takes time $tw = 1/r$ to traverse the link.
**Store-and-Forward Routing**

when a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message.

For store-and-forward routing, the total communication cost for a message of size \( m \) words to traverse \( l \) communication links is:

\[
t_{\text{comm}} = t_s + (m t_w + t_h)l
\]

For parallel platforms using store-and-forward routing, the time is given as:

\[
t_{\text{comm}} = t_s + mlt_w
\]

**Packet Routing**

Store-and-forward routing makes poor use of communication resources. A message is sent from one node to the next only after the entire message has been received. Where as Packet routing is suited to networks with highly dynamic states and higher error rates, such as local- and wide-area networks. This is because individual packets may take different routes and retransmissions can be localized to lost packets.

\[
t_{\text{comm}} = t_s + t_h l + t_w m
\]

Where \( t_w = t_{w1} + t_{w2}(1 + \frac{s}{r}) \)
Figure 2.26. Passing a message from node P0 to P3 (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

In addition to better utilization of communication resources, this principle offers other advantages – lower overhead from packet loss (errors), possibility of packets taking different paths, and better error correction capability.

This technique is the basis for long-haul communication networks such as the Internet, where error rates, number of hops, and variation in network state can be higher.

Cut-Through Routing

The routing scheme resulting from the optimizations is called cut-through routing. In cut-through routing, a message is broken into fixed size units called flow control digits or flits.

Since flits do not contain the overheads of packets, they can be much smaller than packets. A tracer is first sent from the source to the destination node to establish a connection.

Once a connection has been established, the flits are sent one after the other. All flits follow the same path in a dovetailed fashion.

\[ t_{\text{comm}} = t_s + l t_h + t_w m \]
Figure 2.27. An example of deadlock in a cut-through routing network.
2.5.2 Communication Costs in Shared-Address-Space Machines

Associating communication costs with parallel programs is to associate a figure of merit with a program to guide program development.

This task is much more difficult for cache-coherent shared-address-space machines than for message-passing or non-cache-coherent architectures. The reasons are:

• Memory layout is typically determined by the system. The programmer has minimal control on the location of specific data items over and above permuting data structures to optimize access.
• 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. Since cache lines are generally longer than one word, different words might have different access latencies associated with them even for the first access.
• Prefetching can play a role in reducing the overhead associated with data access.
• False sharing is often an important overhead in many programs.
• Contention in shared accesses is often a major contributing overhead in shared address space machines.
2.6 Routing Mechanisms for Interconnection Networks

A routing mechanism determines the path a message takes through the network to get from the source to the destination node.

Routing mechanisms can be classified as minimal or non-minimal. A minimal routing mechanism always selects one of the shortest paths between the source and the destination. A non-minimal routing scheme may route the message along a longer path to avoid network congestion.

One commonly used deterministic minimal routing technique is called dimension-ordered routing. Dimension-ordered routing assigns successive channels for traversal by a message based on a numbering scheme determined by the dimension of the channel.

The dimension ordered routing technique for a two-dimensional mesh is called XY-routing and that for a hypercube is called E-cube routing.

Consider a two-dimensional mesh without wraparound connections. In the XY-routing scheme, a message is sent first along the X dimension until it reaches the column of the destination node and then along the Y dimension until it reaches its destination.
Let $P_{Sy,Sx}$ represent the position of the source node and $P_{Dy,Dx}$ represent that of the destination node. Any minimal routing scheme should return a path of length $|Sx - Dx| + |Sy - Dy|$. Assume that $Dx \geq Sx$ and $Dy \geq Sy$. In the XY-routing scheme, the message is passed through intermediate nodes $P_{Sy,Sx+1}, P_{Sy,Sx+2}, ..., P_{Sy,Dx}$ along the X dimension and then through nodes $P_{Sy+1,Dx}, P_{Sy+2,Dx}, ..., P_{Dy,Dx}$ along the Y dimension to reach the destination. Note that the length of this path is indeed $|Sx - Dx| + |Sy - Dy|$.

Consider a d-dimensional hypercube of $p$ nodes. Let $P_S$ and $P_d$ be the labels of the source and destination nodes. The binary representations of these labels are $d$ bits long. The minimum distance between these nodes is given by the number of ones in $P_S \oplus P_d$ (where $\oplus$ represents the bitwise exclusive-OR operation). In the E-cube algorithm, node $P_S$ computes $P_S \oplus P_d$ and sends the message along dimension $k$, where $k$ is the position of the least significant nonzero bit in $P_S \oplus P_d$. At each intermediate step, node $P_i$, which receives the message, computes $P_i \oplus P_d$ and forwards the message along the dimension corresponding to the least significant nonzero bit. This process continues until the message reaches its destination.
Example 2.16 E-cube routing in a hypercube network

Consider the three-dimensional hypercube shown in Figure 2.28. Let $P_s = 010$ and $P_d = 111$ represent the source and destination nodes for a message. Node $P_s$ computes $010 \oplus 111 = 101$. In the first step, $P_s$ forwards the message along the dimension corresponding to the least significant bit to node 011. Node 011 sends the message along the dimension corresponding to the most significant bit ($011 \oplus 111 = 100$). The message reaches node 111, which is the destination of the message.

Figure 2.28. Routing a message from node $P_s$ (010) to node $P_d$ (111) in a three-dimensional hypercube using E-cube routing.
Properties of Hypercube

- 2 PEs are connected by a direct link iff the binary representation of their labels differ at exactly one bit position.
- In a d-dimensional HyC, each PE is directly connected to d (d = logN) other PEs.
- A d-dimensional HyC can be partitioned into two (d-1)- dimensional sub cubes (fig HC2)
- Since PEs labels have d bits, d such partitions exists.
- Fixing any k – bits in a d- dimensional HC with d – bits. => PEs that differ in the remaining (d-k) bit positions form a (d-k)- dimensional subcube composed of $2^{d-k}$ PEs (such subcubes) (fig HC3)

Eg: $k = 2, \quad d = 4 \quad \Rightarrow$
- 4 subcubes by fixing 2 MSB
- 4 subcubes by fixing 2 LSB
- always 4 subcubes formed by fixing 2 bits
- etc.
- The total number of bit positions at which 2 labels differ = HAMMING distance
  \( s \oplus t \)  Hamming distance
  \( s = \) source
  \( t = \) destination
  \( \oplus = \) EOR (exclusive OR)

=> The number of communication links in the shortest path between 2 PEs is the hamming distance between their labels.

=> The shortest path between any 2 PEs is a HyC, cannot have more than \( d \) links. (Since \( s \oplus t \) cannot contain more than \( d \) bits)

**k-ary d-cubes Nets**

- \( d \)-dimensional HyC is a \((2 \text{ PEs along each link}) \) binary \( d \)-cube or 2-ary \( d \)-cube.
- \( k \)-ary \( d \)-cubes => the number of PEs
- \( d \)-dimension of the network
- \( k \)-radix-ver. of PE in each direction

\[
p = k^d
\]

Eg: 2-d. mesh with \( p \) PEs: \( p = k^2 \) => \( k = \sqrt{p} \)
2.7 Impact of Process-Processor Mapping and Mapping Techniques

A programmer often does not have control over how logical processes are mapped to physical nodes in a network. Even communication patterns that are not inherently congesting may congest the network.

Figure 2.29. Impact of process mapping on performance: (a) underlying architecture; (b) processes and their interactions; (c) an intuitive mapping of processes to nodes; and (d) a random mapping of processes to nodes.
2.7.1 Mapping Techniques for Graphs

Mappings can be used to determine degradation in the performance of an algorithm. Given two graphs, $G(V, E)$ and $G'(V', E')$, mapping graph $G$ into graph $G'$ maps each vertex in the set $V$ onto a vertex (or a set of vertices) in set $V'$ and each edge in the set $E$ onto an edge (or a set of edges) in $E'$. When mapping graph $G(V, E)$ into $G'(V', E')$, three parameters are important. First, it is possible that more than one edge in $E$ is mapped onto a single edge in $E'$. 
The maximum number of edges mapped onto any edge in $E'$ is called the congestion of the mapping. Second, an edge in $E$ may be mapped onto multiple contiguous edges in $E'$.

This is significant because traffic on the corresponding communication link must traverse more than one link, possibly contributing to congestion on the network. The maximum number of links in $E'$ that any edge in $E$ is mapped onto is called the **dilation** of the mapping. Third, the sets $V$ and $V'$ may contain different numbers of vertices. In this case, a node in $V$ corresponds to more than one node in $V'$. 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 2d nodes (labeled 0 through 2d-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.

\[ G(0, 1) = 0 \]

\[ G(1, 1) = 1 \]

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

The function G is called the \textit{binary reflected Gray code} (RGC). The entry G(i, d) denotes the i\textsuperscript{th} entry in the sequence of Gray codes of d bits. Gray codes of d + 1 bits are derived from a table of Gray codes of d bits by reflecting the table and prefixing the reflected entries with a 1 and the original entries with a 0.
Figure 2.30. (a) A three-bit reflected Gray code ring; and (b) its embedding into a three-dimensional hypercube.
Embedding a Mesh into a Hypercube

Embedding a mesh into a hypercube is a natural extension of embedding a ring into a hypercube. We can embed a $2r \times 2s$ wraparound mesh into a $2r+s$-node hypercube by mapping node $(i, j)$ of the mesh onto node $G(i, r-1)||G(j, s-1)$ of the hypercube (where $||$ denotes concatenation of the two Gray codes).
Figure 2.31. (a) A 4 x 4 mesh illustrating the mapping of mesh nodes to the nodes in a four-dimensional hypercube; and (b) a 2 x 4 mesh embedded into a three-dimensional hypercube.
Embedding a Mesh into a Linear Array

Consider first the mapping of a linear array into a mesh. We assume that neither the mesh nor the linear array has wraparound connections.

Figure 2.32. (a) Embedding a 16 node linear array into a 2-D mesh; and (b) the inverse of the mapping. Solid lines correspond to links in the linear array and normal lines to links in the mesh.
Embedding a Hypercube into a 2-D Mesh

Consider the embedding of a p-node hypercube into a p-node 2-D mesh. For the sake of convenience, we assume that p is an even power of two. In this scenario, it is possible to visualize the hypercube as $\sqrt{p}$ sub-cubes, each with $\sqrt{p}$ nodes. Let $d = \log p$ be the dimension of the hypercube. From assumption $d$ is even. Take the $d/2$ least significant bits and use them to define individual subcubes of nodes. For example, in the case of a 4D hypercube, use the lower two bits to define the subcubes as:

$(0000, 0001, 0011, 0010), (0100, 0101, 0111, 0110), (1100, 1101, 1111, 1110),$ and $(1000, 1001, 1011, 1010)$.

The mapping from a hypercube to a mesh can be defined as follows: each $\sqrt{P}$ node subcube is mapped to a $\sqrt{P}$ node row of the mesh. Simply invert the linear array to hypercube mapping to do this. The bisection width of the $\sqrt{P}$ node hypercube is $\sqrt{P}/2$. 
The corresponding bisection width of a $\sqrt{P}$ node row is 1. Therefore the congestion of this subcube-to-row mapping is $\sqrt{P}/2$ (at the edge that connects the two halves of the row). This is illustrated for the cases of $p = 16$ and $p = 32$ in Figure 2.33(a) and (b).

Figure 2.33. Embedding a hypercube into a 2-D mesh.
Process-Processor Mapping and Design of Interconnection Networks

It is possible to map denser networks into sparser networks with associated congestion overheads.

This implies that a sparser network whose link bandwidth is increased to compensate for the congestion can be expected to perform as well as the denser network.

2.7.2 Cost-Performance Tradeoffs

Examine how various cost metrics can be used to investigate cost-performance tradeoffs in interconnection networks. Can analyze by the performance of a mesh and a hypercube network with identical costs.

If the cost of a network is proportional to the number of wires, then a square $p$-node wraparound mesh with $(\log p)/4$ wires per channel costs as much as a $p$-node hypercube with one wire per channel.

For large enough messages, a mesh is always better than a hypercube of the same cost, provided the network is lightly loaded. Even when the network is heavily loaded, the performance of a mesh is similar to that of a hypercube of the same cost.