Chapter 2 Parallel Architectures and Interconnection Networks

> *"The interconnection network is the heart of parallel architecture."* - Chuan-Lin and Tse-Yun Feng [1]

2.1 Introduction

You cannot really design parallel algorithms or programs without an understanding of some of the key properties of various types of parallel architectures and the means by which components can be connected to each other. Parallelism has existed in computers since their inception, at various levels of design. For example, bit-parallel memory has been around since the early 1970s, and simultaneous I/O processing (using channels) has been used since the 1960s. Other forms of parallelism include bit-parallel arithmetic in the arithmetic-logic unit (ALU), instruction look-ahead in the control unit, direct memory access (DMA), data pipe-lining, and instruction pipe-lining. However, the parallelism that we will discuss is at a higher level; in particular we will look at processor arrays, multiprocessors, and multicomputers. We begin, however, by exploring the mathematical concept of a topology

2.2 Network Topologies

We will use the term ***network topology*** to refer to the way in which a set of nodes are connected to each other. In this context, a network topology is essentially a discrete graph – a set of nodes connected by edges. Distance does not exist and there is no notion of the length of an edge[1]. You can think of each edge as being of unit length.

Network topologies arise in the context of parallel architectures as well as in parallel algorithms. In the domain of parallel architectures, network topologies describe the interconnections among multiple processors and memory modules. You will see in subsequent chapters that a network topology can also describe the communication patterns among a set of parallel processes. Because they can be used in these two different ways, we will first examine them purely as mathematical entities, divorced from any particular application.

Formally, a network topology $< S, E >$ is a finite set S of ***nodes*** together with an adjacency relation $E \subseteq S \times S$ on the set. If v and w are nodes such that $(v, w) \in E$, we say that there is a ***directed edge*** from v to w.[2] Sometimes all of the edges in a particular topology will be undirected, meaning that both $(v, w) \in E$ and $(w, v) \in E$. *Unless we state otherwise, we will treat all edges as undirected.* When two nodes are connected by an edge, we say they are ***adjacent***. An example of a network topology that should be quite familiar to the reader is the binary tree shown in Figure 2.1. Notice that the nodes in that figure are ***labeled***. A label is an arbitrary symbol used to refer to the node, nothing more.

We will examine the following topologies in these notes:

- Binary tree
- Fully-connected (also called completely-connected)
- Mesh and torus

[1] Network topologies are a kind of mathematical topological space, for those familiar with this concept, but this is of no importance to us.

[2] The reader familiar with graphs will notice that a network topology is essentially a discrete graph.

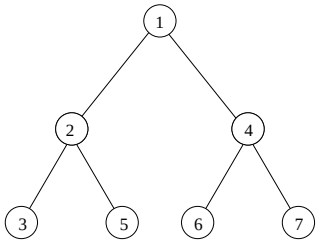

Figure 2.1: Binary tree topology with 7 nodes.

- Hypercube (also called a binary n-cube)
- Butterfly

2.2.1 Network Topology Properties

Network topologies have properties that determine their usefulness for particular applications, which we now define.

Definition 1. A ***path*** from node n_1 to node n_k is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that, for $1 \leq i < k$, n_i is adjacent to n_{i+1}. The length of a path is the number of *edges* in the path, not the number of nodes.

Definition 2. The ***distance*** between a pair of nodes is the length of the *shortest* path between the nodes.

For example, in Figure 2.1, the distance between nodes 3 and 7 is 6, whereas the distance between nodes 3 and 5 is 2.

Definition 3. The ***diameter*** of a network topology is the largest distance between any pair of nodes in the network.

The diameter of the network in Figure 2.1 is 4, since the distance between nodes 3 and 7 is 4, and there is no pair of nodes whose distance is greater than 4. Diameter is important because, if nodes represent processors that must communicate via the edges, which represent communication links, then the diameter determines a lower bound on the communication time. (Note that it is a lower bound and not an upper bound; if a particular algorithm requires, for example, that all pairs of nodes send each other data before the next step of a computation, then the diameter determines how much time will elapse before that step can begin.)

Definition 4. The ***bisection width*** of a network topology is the smallest number of edges that must be deleted to sever the set of nodes into two sets of equal size, or size differing by at most one node.

In Figure 2.1, edge (1,2) can be deleted to split the set of nodes into two sets {2,3,5} and {1,4,6,7}. Therefore, the bisection width of this network is 1. Bisection width is important because it can determine the total communication time. Low bisection width is bad, and high is good. Consider the extreme case, in which a network can be split by removing one edge. This means that all data that flows from one half to the other must pass through this edge. This edge is a bottleneck through which all data must pass sequentially, like a one-lane bridge in the middle of a four-lane highway. In contrast, if the bisection width is high, then many edges must be removed to split the node set. This means that there are many paths from one side of the set to the other, and data can flow in a high degree of parallelism from any one half of the nodes to the other.

Definition 5. The ***degree*** of the network topology is the maximum number of edges that are incident to a node in the topology.

The maximum number of edges per node can affect how well the network scales as the number of processors increases, because of physical limitations on how the network is constructed. A binary tree, for example, has the property that the maximum number of edges per node is 3, regardless of how many nodes are in

the tree. This is good, because the physical design need not change to accommodate the increase in number of processors. Not all topologies have a constant degree. If the degree increases with network size, this generally means that more connections need to be made to each node. Nodes might represent switches, or processors, and in either case they have a fixed pin-out, implying that the connections between processors must be implemented by a complex fan-out of the wires, a very expensive and potentially slow mechanism.

Although the edges in a network topology do not have length, we assume that nodes cannot be infinitely small. As a consequence, the definition of the topology itself can imply that, as the number of nodes increases, the physical distance between them must increase. ***Maximum edge length*** is a measure of this property. It is important because the communication time is a function of how long the signals must travel. It is best if the network can be laid out in three-dimensional space so that the maximum edge length is a constant, independent of network size. If not, and the edge length increases with the number of processors, then communication time increases as the network grows. This implies that expanding the network to accommodate more processors can slow down communication time. The binary tree in Figure 2.1 does not have a constant maximum edge length, because as the size of the tree gets larger, the leaf nodes must be placed further apart, which in turn implies that eventually the edges that leave the root of the tree must get longer.

2.2.2 Binary Tree Network Topology

In a binary tree network, the $2^k - 1$ nodes are arranged in a complete binary tree of depth $k-1$, as in Figure 2.1. The ***depth*** of a binary tree is the length of a path from the root to a leaf node. Each interior node is connected to two children, and each node other than the root is connected to its parent. Thus the degree is 3. The diameter of a binary tree network with $2^k - 1$ nodes is $2(k-1)$, because the longest path in the tree is any path from a leaf node that must go up to the root of the tree and then down to a different leaf node. If we let $n = 2^k - 1$ then $2(k-1)$ is approximately $2\log_2 n$; i.e., the diameter of a binary tree network with n nodes is a logarithmic function of network size, which is very low.

The bisection width is low, which means it is poor. It is possible to split the tree into two sets differing by at most one node in size by deleting either edge incident to the root; the bisection width is 1. As discussed above, maximum edge length is an increasing function of the number of nodes.

2.2.3 Fully-Connected Network Topology

In a ***fully-connected network***, every node is connected to every other node, as in Figure 2.2. If there are n nodes, there will be $n(n-1)/2$ edges. Suppose n is even. Then there are $n/2$ even numbered nodes and $n/2$ odd numbered nodes. If we remove every edge that connects an even node to an odd node, then the even nodes will form a fully-connected network and so will the odd nodes, but the two sets will be disjoint. There are $(n/2)$ edges from each even node to every odd node, so there are $(n/2)^2$ edges that connect these two sets. Not removing any one of them fails to disconnect the two sets, so this is the minimum number. Therefore, the bisection width is $(n/2)^2$. The diameter is 1, since there is a direct link from any node to every other node. The degree is proportional to n, so this network does not scale well. Lastly, the maximum edge length will increase as the network grows, because nodes are not arbitrarily small. (Think of the nodes as lying on the surface of a sphere, and the edges as chords connecting them.)

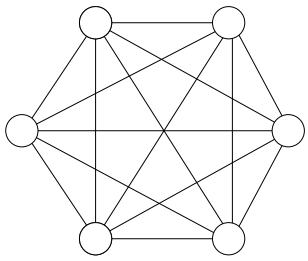

Figure 2.2: Fully-connected network with 6 nodes.

2.2.4 Mesh Network Topology

In a ***mesh network***, nodes are arranged in a q-dimensional lattice. A 2-dimensional lattice with 36 nodes is illustrated in Figure 2.3. The mesh in that figure is square. Unless stated otherwise, meshes are usually square. In general, there are k^2 nodes in a 2-dimensional mesh. A 3-dimensional mesh is the logical extension of a 2-dimensional one. It is not hard to imagine a 3-dimensional mesh. It consists of the lattice points in a 3-dimensional grid, with edges connecting adjacent points. A 3-dimensional mesh, assuming the same number of nodes in all dimensions, must have k^3 nodes. While we cannot visually depict q-dimensional mesh networks when q $>$ 3, we can describe their properties. A q-dimensional mesh network has k^q nodes. k is the number of nodes in a single dimension of the mesh. Henceforth we let q denote the dimension of the mesh.

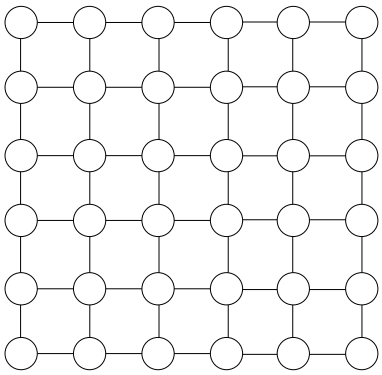

Figure 2.3: A two-dimensional square mesh with 36 nodes.

The diameter of a q-dimensional mesh network with k^q nodes is $q(k-1)$. To see this, note that the farthest distance between nodes is from one corner to the diagonally opposite one. An inductive argument is as follows. In a 2-dimensional lattice with k^2 nodes, you have to travel $(k-1)$ edges horizontally, and $(k-1)$ edges vertically to get to the opposite corner, in any order. Thus you must traverse $2(k-1)$ edges. Suppose we have a mesh of dimension $q-1$, $q > 3$. By assumption its diameter is $(q-1)(k-1)$. A mesh of one higher dimension has $(k-1)$ copies of the $(q-1)$-dimensional mesh, side by side. To get from one corner to the opposite one, you have to travel to the corner of the $(q-1)$-dimensional mesh. That requires crossing $(q-1)(k-1)$ edges, by hypothesis. Then we have to get to the k^{th} copy of the mesh in the new dimension. We have to cross $(k-1)$ more edges to do this. Thus we travel a total of $(q-1)(k-1)+(k-1) = q(k-1)$ edges. This is not rigorous, but this is the idea of the proof.

If k is an even number, the bisection width of a q-dimensional mesh network with k^q nodes is k^{q-1}. Consider the 2D mesh of Figure 2.3. To split it into two halves, you can delete $6 = 6^1$ edges. Imagine the 3D mesh with 216 nodes. To split it into two halves, you can delete the $36 = 6^2$ vertical edges connecting the 36 nodes in the third and fourth planes. In general, one can delete the edges that connect adjacent copies of the $(q-1)$-dimensional lattices in the middle of the q-dimensional lattice. There are k^{q-1} such edges. This is a very high bisection width. One can prove by an induction argument that the bisection width when k is odd is $(k^q-1)/(k-1)$. Thus, whether k is even or odd, the bisection width is $\Theta(k^{q-1})$. As there are $n = k^q$ nodes in the mesh, this is roughly $\sqrt[q]{n}$, which is a very high bisection width. (When $q = 2$, it is $\sqrt{n}$.)

The degree in a mesh is fixed for each given q: it is always 2^q. The maximum edge length is also a constant, independent of the mesh size, for two- and three-dimensional meshes. For higher dimensional meshes, it is not constant.

An extension of a mesh is a ***torus***. A torus, the 2-dimensional version of which is illustrated in Figure 2.4, is an extension of a mesh by the inclusion of edges between the exterior nodes in each row and those in each column. In higher dimensions, it includes edges between the exterior nodes in each dimension. It is called a torus because the surface that would be formed if it were wrapped around the nodes and edges with a thin film would be a mathematical torus, i.e., a doughnut. A torus, or toroidal mesh, has lower diameter than a non-toroidal mesh, by a factor of 2.

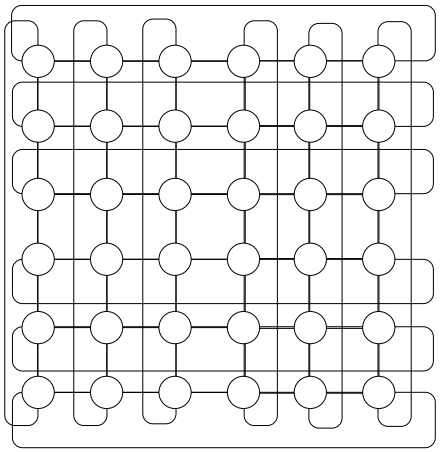

Figure 2.4: Two-dimensional mesh with toroidal connections.

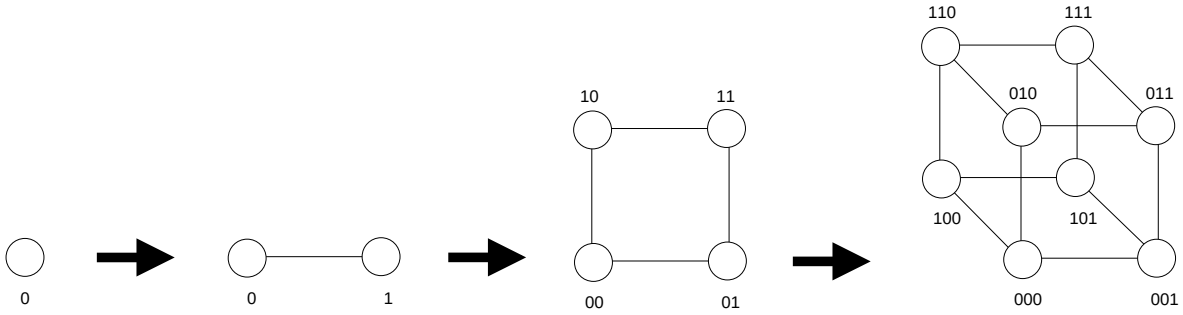

Figure 2.5: Hypercubes of dimensions 0, 1, 2, and 3.

2.2.5 Hypercube (Binary n-Cube)

A ***binary n-cube*** or ***hypercube*** network is a network with 2^n nodes arranged as the vertices of a n-dimensional cube. A hypercube is simply a generalization of an ordinary cube, the three-dimensional shape which you know. Although you probably think of a cube as a rectangular prism whose edges are all equal length, that is not the only way to think about it.

To start, a single point can be thought of as a 0-cube. Suppose its label is 0. Now suppose that we replicate this 0-cube, putting the copy at a distance of one unit away, and connecting the original and the copy by a line segment of length 1, as shown in Figure 2.5. We will give the duplicate node the label, 1.

We extend this idea one step further. We will replicate the 1-cube, putting the copy parallel to the original at a distance of one unit away in an orthogonal direction, and connect corresponding nodes in the copy to those in the original. We will use binary numbers to label the nodes, instead of decimal. The nodes in the copy will be labeled with the same labels as those of the original except for one change: the most significant bit in the original will be changed from 0 to 1 in the copy, as shown in Figure 2.5. Now we repeat this procedure to create a 3-cube: we replicate the 2-cube, putting the copy parallel to the original at a distance of 1 unit away in the orthogonal direction, connect nodes in the copy to the corresponding nodes in the original, and relabel all nodes by adding another significant bit, 0 in the original and 1 in the copy.

It is now not hard to see how we can create hypercubes of arbitrary dimension, though drawing them becomes a bit cumbersome. A 4-cube is illustrated in Figure 2.6 though.

The node labels will play an important role in our understanding of the hypercube. Observe that

- The labels of two nodes differ by exactly one bit change if and only if they are connected by an edge.
- In an n-dimensional hypercube, each node label is represented by n bits. Each of these bits can be

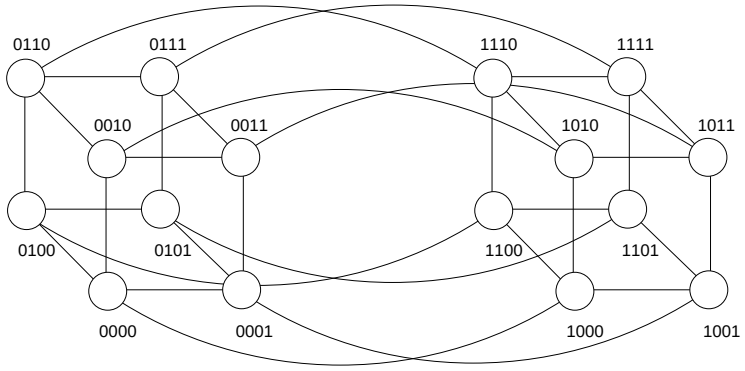

Figure 2.6: A 4-dimensional hypercube.

inverted (0->1 or 1->0), implying that each node has exactly n incident edges. In the 4D hypercube, for example, each node has 4 neighbors. Thus the degree of an n-cube is n.

- The diameter of an n-dimensional hypercube is n. To see this, observe that a given integer represented with n bits can be transformed to any other n-bit integer by changing at most n bits, one bit at a time. This corresponds to a walk across n edges in a hypercube from the first to the second label.
- The bisection width of an n-dimensional hypercube is 2^{n-1}. One way to see this is to realize that all nodes can be thought of as lying in one of two planes: pick any bit position and call it b. The nodes whose b-bit = 0 are in one plane, and those whose b-bit = 1 are in the other. To split the network into two sets of nodes, one in each plane, one has to delete the edges connecting the two planes. Every node in the 0-plane is attached to exactly one node in the 1-plane by one edge. There are 2^{n-1} such pairs of nodes, and hence 2^{n-1} edges. No smaller set of edges can be cut to split the node set.
- The number of edges in an n-dimensional hypercube is $n \cdot 2^{n-1}$. To see this, note that it is true when $n = 0$, as there are 0 edges in the 0-cube. Assume it is true for all $k < n$. A hypercube of dimension n consists of two hypercubes of dimension $n-1$ with one edge between each pair of corresponding nodes in the two smaller hypercubes. There are 2^{n-1} such edges. Thus, using the inductive hypothesis, the hypercube of dimension n has $2 \cdot (n-1) \cdot 2^{n-2} + 2^{n-1} = (n-1) \cdot 2^{n-1} + 2^{n-1} = (n-1+1) \cdot 2^{n-1} = n \cdot 2^{n-1}$ edges. By the axiom of induction, it is proved.

The bisection width is very high (one half the number of nodes), and the diameter is low. This makes the hypercube an attractive organization. Its primary drawbacks are that (1) the number of edges per node is a (logarithmic) function of network size, making it difficult to scale up, and the maximum edge length increases as network size increases.

2.2.6 Butterfly Network Topology

A ***butterfly network topology*** consists of $(k+1)2^k$ nodes arranged in $k+1$ ranks, each containing $n = 2^k$ nodes. k is called the ***order*** of the network. The ranks are labeled 0 through k. Figure 2.7 depicts a butterfly network of order 3, meaning that it has 4 ranks with $2^3 = 8$ nodes in each rank. The columns in the figure are labeled 0 through 7.

We describe two different methods for constructing a butterfly network of order k.

Method 1:

- Create $k+1$ ranks labeled 0 through k, each containing 2^k nodes, labeled 0 through $2^k - 1$.
- Let $[i, j]$ denote the node in rank i, column j.

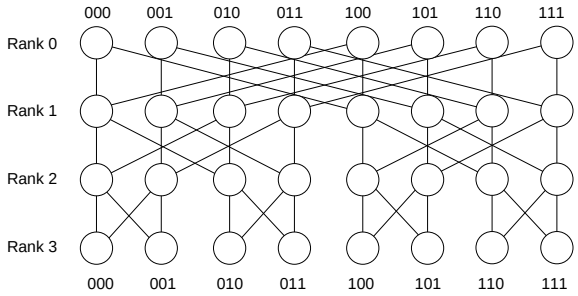

Figure 2.7: Butterfly network of order 3.

- For each rank, from 0 through $k-1$, connect all nodes $[i, j]$ to nodes $[i+1, j]$. In other words, draw the straight lines down the columns as shown in Figure 2.7.
- For each rank, from 0 through $k-1$, connect each node $[i, j]$ to node $[i+1$, $(j+2^{k-i-1}$. This creates the diagonal edges that form the butterfly pattern. For example, if $k = 3$, in rank 0, the node in column j is connected to the node in rank 1 in column $(j+2^{k-1})\%2^k = (j+4)\%8$, so the nodes 0,1,2, and 3 in rank 0 are connected to the nodes 4,5,6, and 7 respectively in rank 1, and nodes 4,5,6, and 7 in rank 0 are connected to nodes 0,1,2, and 3 respectively in rank 1.

Method 2: This is a recursive definition of a butterfly network.

- A butterfly network of order 0 consists of a single node, labeled $[0, 0]$.
- To form a butterfly network of order $k+1$, replicate the butterfly network of order k, labeling the corresponding nodes in the copy by adding 2^k to their column numbers. Place the copy to the right of the original so that nodes remain in the same ranks. Add one to all rank numbers and create a new rank 0. For each $j = 0, 1, , ..., 2^{k+1}$, connect the node in column j of the new rank 0 to two nodes: the node in rank 1, column$j$ and the corresponding node in the copy.

This recursive sequence is illustrated in Figures 2.8 and 2.9.

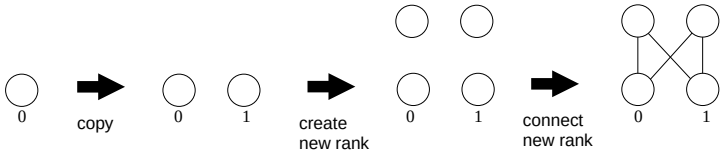

Figure 2.8: Recursive construction of butterfly network of order 1 from order 0.

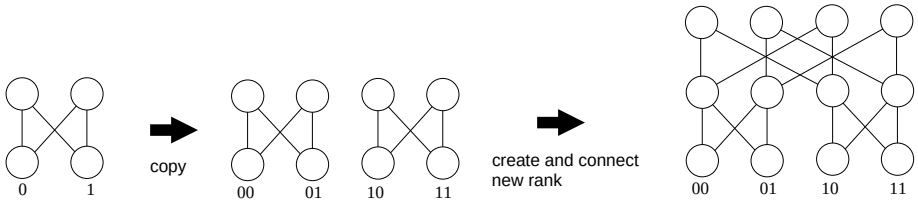

Figure 2.9: Recursive construction of butterfly network of order 2 from order 1.

It can be shown that there is a path from any node in the first rank to any node in the last rank. If this is the case, then the diameter is $2k$: to get from node 0 to node 7 in the first rank requires descending to rank 3 and returning along a different path. If, however, the last rank is really the same as the first rank, which is

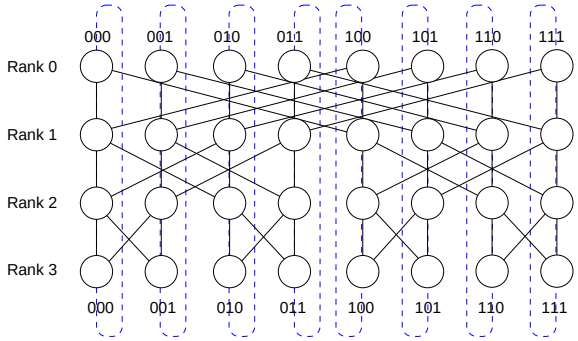

Figure 2.10: Butterfly network with first and last ranks representing the same node.

sometimes the case, then the diameter is k. Figure 2.10 schematically represents this type of network with dashed lines between the first and last ranks.

The bisection width is 2^k. To split the network requires deleting all edges that cross between columns $2^{k-1} - 1$ and $2^k - 1$. Only the nodes in rank 0 have connections that cross this divide. There are 2^k nodes in rank 0, and one edge from each to a node on the other side of the imaginary dividing line. This network topology has a fixed number of edges per node, 4 in total; however the maximum edge length will increase as the order of the network increases.

One last observation: in Figure 2.7, imagine taking each column and enclosing it in a box. Call each box a node. There are 2^k such nodes. Any edge that was incident to any node within the box is now considered incident to the new node (i.e., the box). The resulting network contains 2^k nodes connected in a k-dimensional hypercube. This is the relationship between butterfly and hypercube networks.

2.3 Interconnection Networks

We now focus on actual, physical connections. An ***interconnection network*** is a system of links that connects one or more devices to each other for the purpose of inter-device communication. In the context of computer architecture, an interconnection network is used primarily to connect processors to processors, or to allow multiple processors to access one or more shared memory modules. Sometimes they are used to connect processors with locally attached memories to each other. The way that these entities are connected to each other has a significant effect on the cost, applicability, scalability, reliability, and performance of a parallel computer. In general, the entities that are connected to each other, whether they are processors or memories, will be called ***nodes***.

An interconnection network may be classified as shared or switched. A ***shared network*** can have at most one message on it at any time. For example, a ***bus*** is a shared network, as is traditional Ethernet. In contrast, a ***switched network*** allows point-to-point messages among pairs of nodes and therefore supports the transfer of multiple concurrent messages. Switched Ethernet is, as the name implies, a switched network. Shared networks are inferior to switched networks in terms of performance and scalability. Figure 2.11 illustrates a shared network. In the figure, one node is sending a message to another, and no other message can be on the network until that communication is completed.

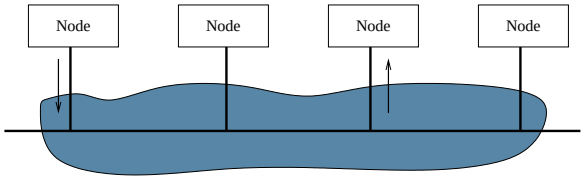

Figure 2.11: A shared network connecting 4 nodes.

Figure 2.12 depicts a switched network in which two simultaneous connections are taking place, indicated by the dashed lines.

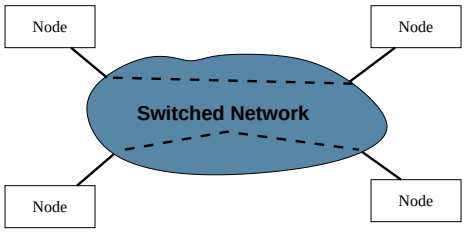

Figure 2.12: A switched network connecting 4 nodes.

2.3.1 Interconnection Network Topologies

Notation. When we depict networks in these notes, we will follow the same convention as the Quinn book and use squares to represent processors and/or memories, and circles to represent switches. For example, in Figure 2.13, there are four processors on the bottom row and a binary tree of seven switches connecting them.

We distinguish between ***direct*** and ***indirect*** topologies. In a direct topology, there is exactly one switch for each processor node, whereas in an indirect topology, the number of switches is greater than the number of processor nodes. In Figure 2.13, the topology is indirect because there are more switches than processor nodes.

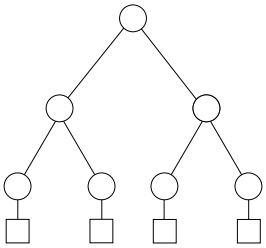

Figure 2.13: Binary tree interconnection network. The circles are switches and the squares are processors.

Certain topologies are usually used as direct topologies, others as indirect topologies. In particular,

- The 2D mesh is almost always used as a direct topology, with a processor attached to each switch, as shown in Figure 2.14.
- Binary trees are always indirect topologies, acting as a switching network to connect a bank of processors to each other, as shown in Figure 2.13.
- Butterfly networks are always indirect topologies; the processors are connected to rank 0, and either memory modules or switches back to the processors are connected to the last rank.
- Hypercube networks are always direct topologies.

2.4 Vector Processors and Processor Arrays

There are two essentially different models of parallel computers: vector computers and multiprocessors. A ***vector computer*** is simply a computer that has an instruction that can operate on a vector. A ***pipelined***

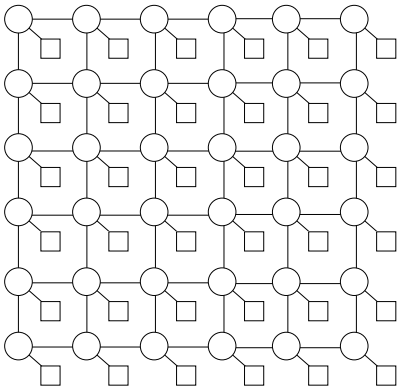

Figure 2.14: 2D mesh interconnection network, with processors (squares) attached to each switch.

vector processor is a vector processor that can issue a vector instruction that operates on all of the elements of the vector in parallel by sending those elements through a highly pipelined functional unit with a fast clock. A ***processor array*** is a vector processor that achieves parallelism by having a collection of identical, synchronized ***processing elements*** (PE), each of which executes the same instruction on different data, and which are controlled by a single control unit. Because all PEs execute the same instruction at the same time, this type of architecture is suited to problems with data parallelism. Processor arrays are often used in scientific applications because these often contain a high amount of data parallelism.

2.4.1 Processor Array Architecture

In a processor array, each PE has a unique identifier, its processor id, which can be used during the computation. Each PE has a small, local memory in which its own private data can be stored. The data on which each PE operates is distributed among the PEs' local memories at the start of the computation, provided it fits in that memory. The control unit, which might be a full-fledged CPU, broadcasts the instruction to be executed to the PEs, which execute it on data from its local memory, and can store the result in their local memories or can return global results back to the CPU. A global result line is usually a separate, parallel bus that allows each PE to transmit values back to the CPU to be combined by a parallel, global operation, such as a logical-and or a logical-or, depending upon the hardware support in the CPU. Figure 2.15 contains a schematic diagram of a typical processor array. The PEs are connected to each other through an interconnection network that allows them to exchange data with each other as well.

If the processor array has N PEs, then the time it takes to perform the same operation on N elements is the same as to perform it on one element. If the number of data items to be manipulated is larger than N, then it is usually the programmer's job to arrange for the additional elements to be stored in the PEs' local memories and operated on in the appropriate sequence. For example, if the machine has 1024 PEs and an array of size 5000 must be processed, then since $5000 = 4 \cdot 1024 + 4$, the 5000 elements would be distributed among the PEs by giving 4 elements to 1020 PEs and 5 to 4 PEs.

The topology of the interconnection network determines how easy it is to perform different types of computations. If the interconnection network is a butterfly network, for example, then messages between PEs will travel many links in each communication, slowing down the computation greatly. If the machine is designed for fast manipulation of two-dimensional data sets, such as images or matrices, then the interconnection network would be a 2D mesh arranged as a direct topology, as shown in Figure 2.14.

2.4.2 Processor Array Performance

Although the subject of parallel algorithm performance will be covered in depth in a later chapter, we introduce a measure of ***computer performance*** here, namely the number of operations per second that a computer can execute. There are many other metrics by which a computer's performance can be evaluated; this is just one common one.

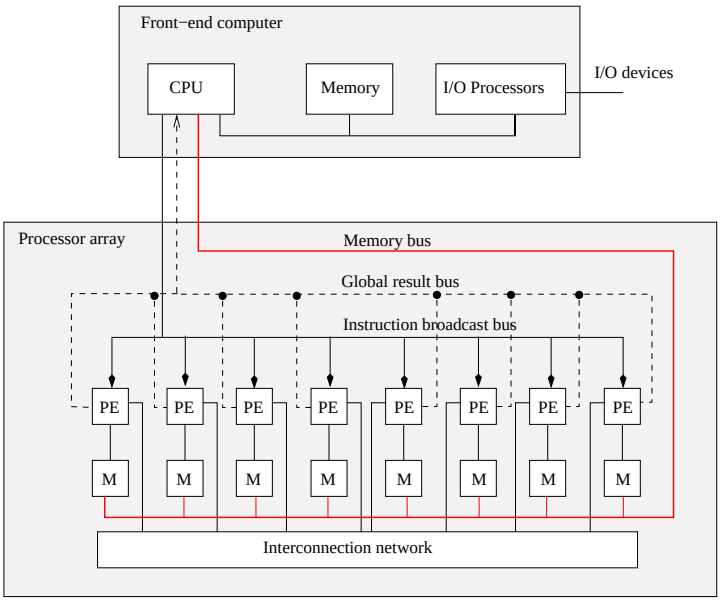

Figure 2.15: Typical processor array architecture.

If a processor array has 100 processing elements, and each is kept busy all of the time and executes H operations per second, then the processor array executes $100H$ operations per second. If, at any moment in time, on average, only a fraction f of the PEs are active, then the performance is reduced to $100fH$ operations per second. The average fraction of PEs that are actively executing instructions at any time is called the utilization of the processor array.

Several factors influence this utilization. One, as alluded to above, is whether or not the number of data elements to be processed is a multiple of the number of PEs. If it is not, then at some point there will be idle PEs while others are active.

Example 6. A processor array has 512 PEs. Two arrays `A` and `B` of 512 floating point numbers each is to be added and stored into a third array `C`. The PEs can execute a floating point addition (including fetching and storing) in 100 nanoseconds. The performance of this processor array on this problem would be

$$\frac{512\ operations}{100\ nanosecs} = \frac{512\ operations}{100 \times 10^{-9}\ seconds} = 5.12 \times 10^9\ flops.$$

The term ***flops*** is short for ***floating-point operations per second***.

Suppose the size of the array is 700, instead. In this case the first 512 elements will be executed in parallel by all PEs, taking 100 nanoseconds in total, but the remaining 188 elements will require only 188 PEs to be active, and 100 nanoseconds will elapse. Therefore the performance will be

$$\frac{700\ operations}{200\ nanosecs} = \frac{3.5\ operations}{10^{-9}\ seconds} = 3.5 \times 10^9\ flops.$$

This is less than 70% of peak performance.

2.4.3 Processor Masking

A processor array also has to handle what PEs do when they do not need to participate in a computation, as in the preceding example. This problem also arises when conditional instructions such as if-then-else instructions are executed. In this case, it is possible that the true-branch will need to be taken in some PEs and the false-branch in others. The problem is that the control unit issues a single instruction to all PEs –

there is no mode in which some PEs execute one instruction and others, a different one. Therefore, the way a processor array has to work is that when the true branch is executed, the PEs that are not supposed to execute that instruction must be de-activated, and when the false branch is executed, the ones that are not supposed to execute that instruction must be de-activated.

This de-activation is achieved by giving the PEs a mask bit, which they can set and unset. When a condition is tested by a PE, if it must be de-activated in the next instruction, it sets its mask bit to prevent it from executing. After the instruction sequence for the true branch is complete, the control unit issues an instruction to each PE to invert its mask bit. The ones that were masked out are unmasked, and the ones that had been active become inactive because their mask bits are set. Then the false branch instructions are executed and the control unit sends an instruction to all PEs to clear their mask bits. It should be apparent that when a program contains a large fraction of conditional-executed code, the fraction of idle PEs increases and the performance decreases.

2.4.4 Summary

- Processor arrays are suitable for highly data parallel problems, but not those that have little data parallelism.
- When a program has a large fraction of conditionally-executed code, a processor array will not perform well.
- Processor arrays are designed to solve a single problem at a time, running to completion, as in batch style processing, because context switching is too costly on them.
- The cost of the front end of a processor array and the interconnection network is high and this cost must be amortized over a large number of PEs to make it cost effective. For this reason, processor arrays are most cost-effective when the number of PEs is very large.
- One of the primary reasons that processor arrays became popular was that the control unit of a processor was costly to build. As control units have become less expensive, processor arrays have become less competitively priced in comparison to multicomputers.

2.5 Multiprocessors

In keeping with Quinn's usage[2], we will use the term ***multiprocessor*** to mean a computer with multiple CPUs and a shared memory. (This is what many others call a shared memory multiprocessor.) In a multiprocessor, the same address generated on two different CPUs refers to the same memory location. Multiprocessors are divided into two types: those in which the shared memory is physically in one place, and one in which it is distributed among the processors.

2.5.1 Centralized (Shared Memory) Multiprocessors

In a centralized memory multiprocessor, all processors have equal access to the physical memory. This type of multicomputer is also called a ***uniform memory access*** (***UMA***) ***multiprocessor***. A UMA multiprocessor may also be called a ***symmetric multiprocessor,*** or ***SMP***. UMA multiprocessors are relatively easy to build because additional processors can be added to the bus of a conventional uniprocessor machine. Because modern caches greatly reduce the need for primary memory accesses, the increase in bus traffic does not become a major performance issue until the number of CPUs is more than a few dozen or so.

Because there is a shared physical memory, independent processes running on separate CPUs can share data; it becomes the programmer's responsibility to ensure that the data is accessed without race conditions; usually the hardware in this type of machine provides various instructions to make this easier for the programmer, such as barrier synchronization primitives, or semaphore operations. A ***barrier synchronization*** instruction is an instruction that, when it is executed by a process, causes that process to wait until all other

cooperating processes have reached that same instruction in their code. This will be described in more detail in a later chapter. Semaphores, and semaphore operations, are also a means to allow processes to cooperate in how they access a region of their code that accesses a shared data item, but these too require that the programmer use them correctly. The processes can also have their own private data, which is data used only by a single process. Figure 2.16 depicts a UMA multiprocessor with four CPUs.

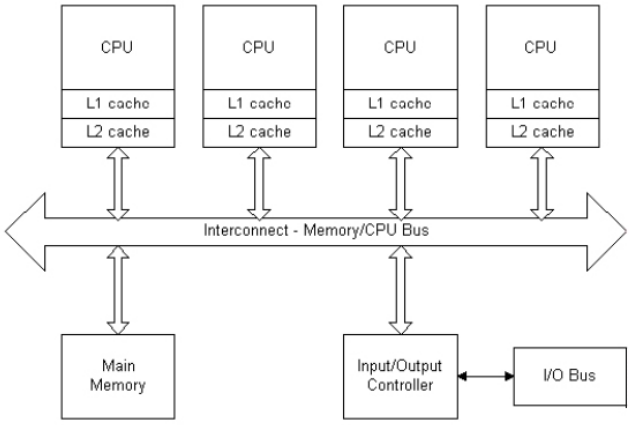

Figure 2.16: UMA multiprocessor.

The hardware designer must address the ***cache coherence problem***. In a shared memory multiprocessor with separate caches in each processor, the problem is that the separate caches can have copies of the same memory blocks, and unless measures are taken to prevent it, the copies will end up having different values for the blocks. This will happen whenever two different processors modify their copies with different values and nothing is done to propagate the changes to other caches. Usually, a ***snooping cache protocol*** is used in these types of machines. We will not explain it here.

2.5.2 Distributed (Shared Memory) Multiprocessors

In a centralized memory multiprocessor, the shared access to a common memory through a bus limits how many CPUs can be accommodated. The alternative is to attach separate memory modules to each processor. When all processors can access the memory modules attached to all other processors, it is called a ***distributed multiprocessor***, or a ***non-uniform memory access (NUMA) multiprocessor***. Because it is much faster to access the local memory attached directly to the processor than the modules attached to other processors, the access is not uniform, hence the name. Executing programs tend to obey the principles of ***spatial locality*** and ***temporal locality***. Spatial locality means that the memory locations that they access tend to be near each other in small windows of time, and temporal locality means that memory locations that are accessed once tend to be accessed frequently in a small window of time. This behavior can be used to advantage so that most of the process's references to memory are in its private memory.

When shared memory is distributed in this way, the cache coherence problem cannot be solved with snooping protocols because the "snooping" becomes inefficient for large numbers of processors. On these machines, a ***directory-based protocol*** is used. We will not describe that protocol here. The interested reader is referred to the Quinn book [2].

2.6 Multicomputers

A ***multicomputer*** is a distributed memory, multiple-CPU computer, but the memory is not shared. Each CPU has its own address space and can access only its own local memory, which is called private memory.

Thus, the same address on two different CPUs refers to two different memory locations. These machines are also called ***private-memory multiprocessors***. Because there is no shared address space, the only way for processes running on different CPUs to communicate is through some type of message-passing system, and the architecture of this type of multicomputer typically supports efficient message-passing.

A ***commercial multicomputer*** is a multicomputer designed, manufactured, and intended to be sold as a multicomputer. A ***commodity cluster*** is a multicomputer put together out of off-the-shelf components to create a multicomputer. A commercial multicomputer's interconnection network and processors are optimized to work with each other, providing low-latency, high-bandwidth connections between the computers, at a higher price tag than a commodity cluster. Commodity clusters, though, generally have lower performance, with higher latency and lower bandwidth in the interprocessor connections.

2.6.1 Asymmetrical Multicomputers

Some multicomputers are designed with a special front-end computer and back-end computers, the idea being that the front-end serves as the master and gateway for the machine, and the back-end CPUs are for computation. These types of machines are called ***asymmetrical multicomputers***. Users login to the front-end and their jobs are run on the back-end processors, and all I/O takes place through the front-end machine. The software that runs on the front and back ends is also different. The front-end has a more powerful and versatile operating system and compilers, whereas the back-end processors have scaled down operating systems that require less memory and resources. See Figure 2.17

The problems with this design are that:

- the front-end is a bottleneck;
- it does not scale well because of the front-end, as the performance of the front-end host limits the number of users and the number of jobs;
- the pared-down operating systems on the back-end do not allow for sophisticated debugging tools;
- the parallel programs must be written in two parts – the part that runs on the front-end and interacts with the I/O devices and the user, and the part that runs on the back-end.

These last two problems were such an impediment that many asymmetrical multicomputers include advanced debugging facilities and I/O support on the back-end hosts.

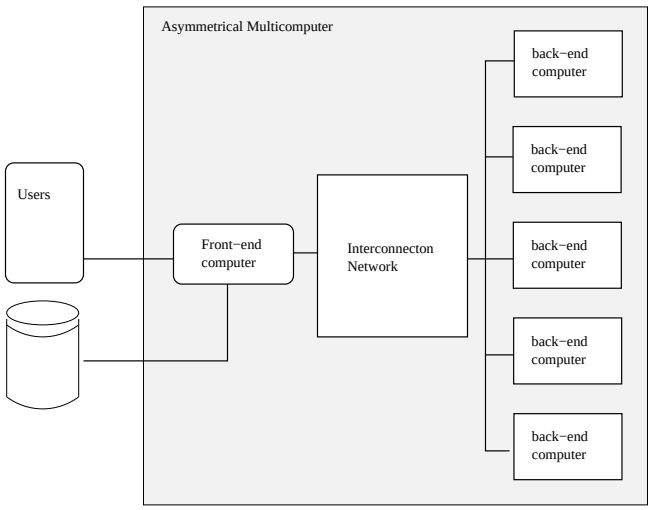

Figure 2.17: Asymmetrical multicomputer.

2.6.2 Symmetrical Multicomputers

A ***symmetrical multicomputer*** is one in which all of the hosts are identical and are connected to each other through an interconnection network. Users can login to any host and the file system and I/O devices are equally accessible from every host. This overcomes the bottleneck of a front-end host as well as the scalability issue, but it has many other problems. It is not really a suitable environment for running large scale parallel programs, because there is little control over how many jobs are running on any one host, and all hosts are designed to allow program development and general interactive use, which degrades performance of a highly compute-bound job. See Figure 2.18.

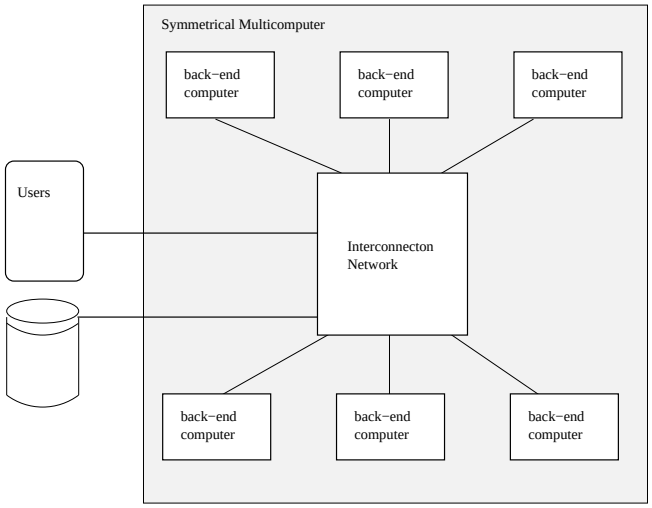

Figure 2.18: Symmetrical multicomputer.

2.7 Flynn's Taxonomy

In the 1966, Michael Flynn [3] proposed a categorization of parallel hardware based upon a classification scheme with two orthogonal parameters: the instruction stream and the data stream. In his taxonomy, a machine was classified by whether it has a single or multiple instruction streams, and whether it had single or multiple data streams. An instruction stream is a sequence of instructions that flow through a processor, and a data stream is a sequence of data items that are computed on by a processor. For example, the ordinary single-processor computer has a single instruction stream and a single data stream.

This scheme leads to four independent types of computer:

SISD — single instruction, single data; i.e., a conventional uni-processor.

SIMD — single instruction, multiple data; like MMX or SSE instructions in the x86 processor series, processor arrays and pipelined vector processors. SIMD multiprocessors issue a single instruction that operates on multiple data items simultaneously. Vector processors are SIMD multiprocessors, which means that processor arrays and pipelined vector processors are also SIMD machines.

MISD — multiple instruction, single data; very rare but one example is the U.S. Space Shuttle flight controller. Systolic arrays fall into this category as well.

MIMD — multiple instruction, multiple data; SMPs, clusters. This is the most common multiprocessor. MIMD multiprocessors are more complex and expensive, and so the number of processors tends to be smaller than in SIMD machines. Today's multiprocessors are found on desktops, with anywhere from 2 to 16 processors. Multiprocessors and multicomputers fall into this category.

References

[1] C. Wu and T. Feng. *Tutorial, interconnection networks for parallel and distributed processing.* Tutorial Texts Series. IEEE Computer Society Press, 1984.

[2] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

[3] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

Subject Index