2.5 Communication Costs in Parallel Machines

One of the major overheads in the execution of parallel programs arises from communication of information between processing elements. 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. These issues form the focus of our discussion here.

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:

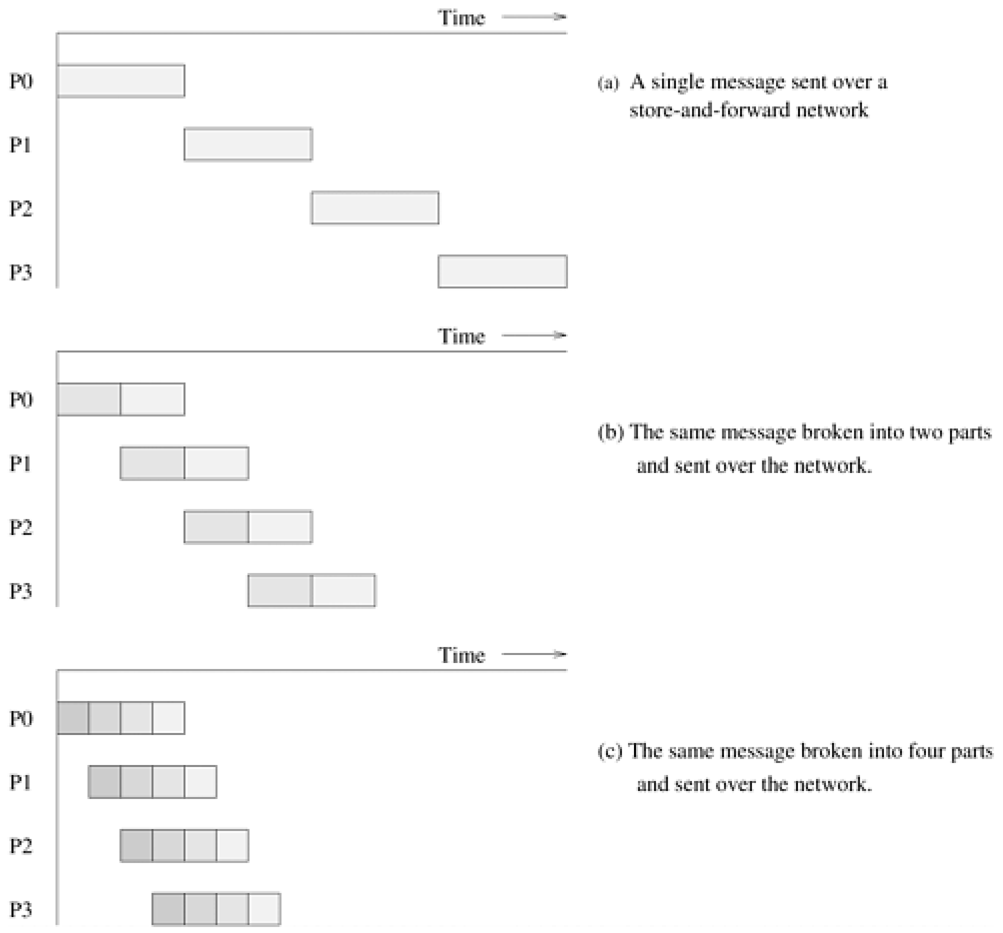
1. *Startup time* (t_s): The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local node and the router. This delay is incurred only once for a single message transfer.
2. *Per-hop time* (t_h): After a message leaves a node, it takes a finite amount of time to reach the next node in its path. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. It is also known as *node latency*. The per-hop time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.
3. *Per-word transfer time* (t_w): If the channel bandwidth is r words per second, then each word takes time $t_w = 1/r$ to traverse the link. This time is called the per-word transfer time. This time includes network as well as buffering overheads.

We now discuss two routing techniques that have been used in parallel computers – store-and-forward routing and cut-through routing.

Store-and-Forward Routing

In 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. Figure 2.26(a) shows the communication of a message through a store-and-forward network.

Figure 2.26. Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.


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

Suppose that a message of size *m* is being transmitted through such a network. Assume that it traverses *l* links. At each link, the message incurs a cost t_h for the header and $t_w m$ for the rest of the message to traverse the link. Since there are *l* such links, the total time is $(t_h + t_w m)l$. Therefore, for store-and-forward routing, the total communication cost for a message of size *m* words to traverse *l* communication links is

Equation 2.2

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

In current parallel computers, the per-hop time t_h is quite small. For most parallel algorithms, it is less than $t_w m$ even for small values of *m* and thus can be ignored. For parallel platforms using store-and-forward routing, the time given by Equation 2.2 can be simplified to

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

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 (Figure 2.26(a)). Consider the scenario shown in Figure 2.26(b), in which the original message is broken into two equal sized parts before it is sent. In this case, an intermediate node waits for only half of the original message to arrive before passing it on. The increased utilization of communication resources and reduced communication time is apparent from Figure 2.26(b). Figure 2.26(c) goes a step further and breaks the message into four parts. 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. For these reasons, 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. Of course, the overhead here is that each packet must carry routing, error correction, and sequencing information.

Consider the transfer of an *m* word message through the network. The time taken for programming the network interfaces and computing the routing information, etc., is independent of the message length. This is aggregated into the startup time t_s of the message transfer. We assume a scenario in which routing tables are static over the time of message transfer (i.e., all packets traverse the same path). While this is not a valid assumption under all circumstances, it serves the purpose of motivating a cost model for message transfer. The message is broken into packets, and packets are assembled with their error, routing, and sequencing fields. The size of a packet is now given by $r + s$, where r is the original message and s is the additional information carried in the packet. The time for packetizing the message is proportional to the length of the message. We denote this time by mt_{w1}. If the network is capable of communicating one word every t_{w2} seconds, incurs a delay of t_h on each hop, and if the first packet traverses l hops, then this packet takes time $t_h l + t_{w2}(r + s)$ to reach the destination. After this time, the destination node receives an additional packet every $t_{w2}(r + s)$ seconds. Since there are $m/r - 1$ additional packets, the total communication time is given by:

$$\begin{aligned} t_{comm} &= t_s + t_{w1}m + t_h l + t_{w2}(r+s) + \left(\frac{m}{r} - 1\right) t_{w2}(r+s) \\ &= t_s + t_{w1}m + t_h l + t_{w2}m + t_{w2}\frac{s}{r}m \\ &= t_s + t_h l + t_w m, \end{aligned}$$

where

$$t_w = t_{w1} + t_{w2}\left(1 + \frac{s}{r}\right).$$

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.

Cut-Through Routing

In interconnection networks for parallel computers, additional restrictions can be imposed on message transfers to further reduce the overheads associated with packet switching. By forcing all packets to take the same path, we can eliminate the overhead of transmitting routing information with each packet. By forcing in-sequence delivery, sequencing information can be eliminated. By associating error information at message level rather than packet level, the overhead associated with error detection and correction can be reduced. Finally, since error rates in interconnection networks for parallel machines are extremely low, lean error detection mechanisms can be used instead of expensive error correction schemes.

The routing scheme resulting from these 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. An intermediate node does not wait for the entire message to arrive before forwarding it. As soon as a flit is received at an intermediate node, the flit is passed on to the next node. Unlike store-and-forward routing, it is no longer necessary to have buffer space at each intermediate node to store the entire message. Therefore, cut-through routing uses less memory and memory bandwidth at intermediate nodes, and is faster.

Consider a message that is traversing such a network. If the message traverses l links, and t_h is the per-hop time, then the header of the message takes time lt_h to reach the destination. If the message is m words long, then the entire message arrives in time $t_w m$ after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is

Equation 2.3

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

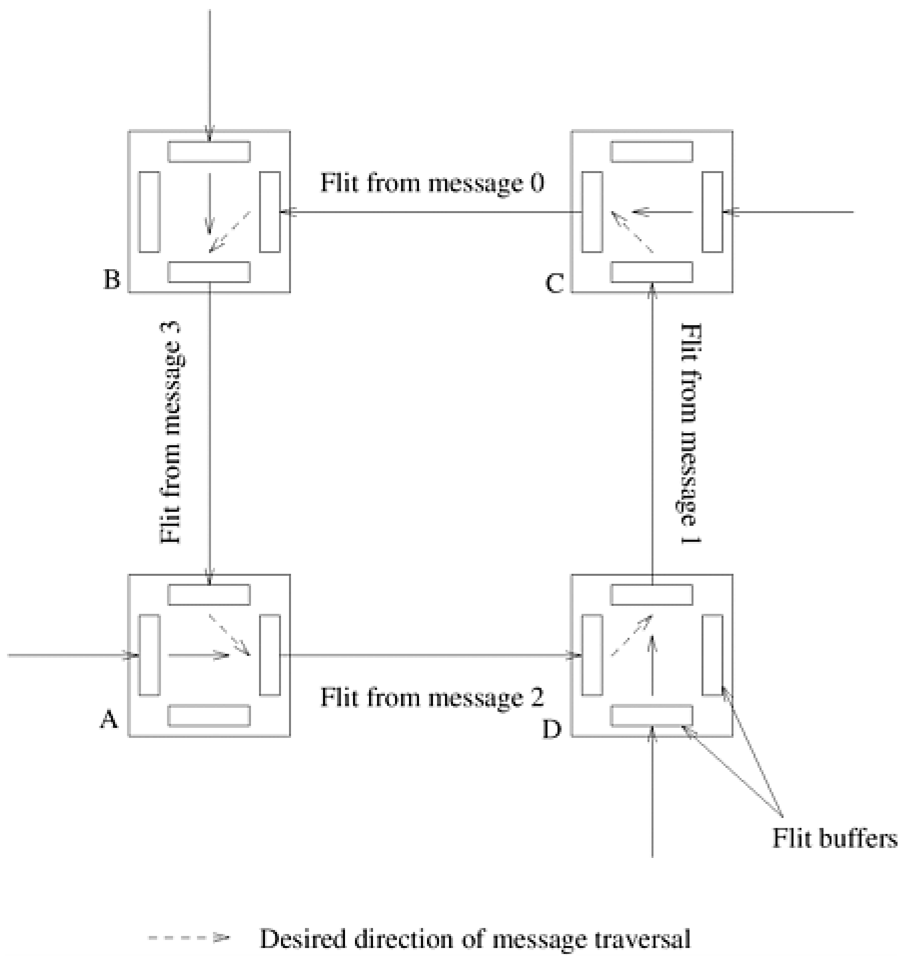
This time is an improvement over store-and-forward routing since terms corresponding to number of hops and number of words are additive as opposed to multiplicative in the former. Note that if the communication is between nearest neighbors (that is, $l = 1$), or if the message size is small, then the communication time is similar for store-and-forward and cut-through routing schemes.

Most current parallel computers and many local area networks support cut-through routing. The size of a flit is determined by a variety of network parameters. The control circuitry must operate at the flit rate. Therefore, if we select a very small flit size, for a given link bandwidth, the required flit rate becomes large. This poses considerable challenges for designing routers as it requires the control circuitry to operate at a very high speed. On the other hand, as flit sizes become large, internal buffer sizes increase, so does the latency of message transfer. Both of these are undesirable. Flit sizes in recent cut-through interconnection networks range from four bits to 32 bytes. In many parallel programming paradigms that rely predominantly on short messages (such as cache lines), the latency of messages is critical. For these, it is unreasonable for a long message traversing a link to hold up a short message. Such scenarios are addressed in routers using multilane cut-through routing. In multilane cut-through routing, a single physical channel is split into a number of virtual channels.

Messaging constants t_s, t_w, and t_h are determined by hardware characteristics, software layers, and messaging semantics. Messaging semantics associated with paradigms such as message passing are best served by variable length messages, others by fixed length short messages. While effective bandwidth may be critical for the former, reducing latency is more important for the latter. Messaging layers for these paradigms are tuned to reflect these requirements.

While traversing the network, if a message needs to use a link that is currently in use, then the message is blocked. This may lead to deadlock. Figure 2.27 illustrates deadlock in a cut-through routing network. The destinations of messages 0, 1, 2, and 3 are *A*, *B*, *C*, and *D*, respectively. A flit from message 0 occupies the link *CB* (and the associated buffers). However, since link *BA* is occupied by a flit from message 3, the flit from message 0 is blocked. Similarly, the flit from message 3 is blocked since link *AD* is in use. We can see that no messages can progress in the network and the network is deadlocked. Deadlocks can be avoided in cut-through networks by using appropriate routing techniques and message buffers. These are discussed in Section 2.6.

Figure 2.27. An example of deadlock in a cut-through routing network.



A Simplified Cost Model for Communicating Messages

As we have just seen in Section 2.5.1, the cost of communicating a message between two nodes *l* hops away using cut-through routing is given by

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

This equation implies that in order to optimize the cost of message transfers, we would need to:

1. Communicate in bulk. That is, instead of sending small messages and paying a startup cost t_s for each, we want to aggregate small messages into a single large message and amortize the startup latency across a larger message. This is because on typical platforms such as clusters and message-passing machines, the value of t_s is much larger than those of t_h or t_w.
2. Minimize the volume of data. To minimize the overhead paid in terms of per-word transfer time t_w, it is desirable to reduce the volume of data communicated as much as

possible.

3. Minimize distance of data transfer. Minimize the number of hops l that a message must traverse.

While the first two objectives are relatively easy to achieve, the task of minimizing distance of communicating nodes is difficult, and in many cases an unnecessary burden on the algorithm designer. This is a direct consequence of the following characteristics of parallel platforms and paradigms:

- In many message-passing libraries such as MPI, the programmer has little control on the mapping of processes onto physical processors. In such paradigms, while tasks might have well defined topologies and may communicate only among neighbors in the task topology, the mapping of processes to nodes might destroy this structure.
- Many architectures rely on randomized (two-step) routing, in which a message is first sent to a random node from source and from this intermediate node to the destination. This alleviates hot-spots and contention on the network. Minimizing number of hops in a randomized routing network yields no benefits.
- The per-hop time (t_h) is typically dominated either by the startup latency (t_s)for small messages or by per-word component ($t_w m$) for large messages. Since the maximum number of hops (l) in most networks is relatively small, the per-hop time can be ignored with little loss in accuracy.

All of these point to a simpler cost model in which the cost of transferring a message between two nodes on a network is given by:

Equation 2.4

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

This expression has significant implications for architecture-independent algorithm design as well as for the accuracy of runtime predictions. Since this cost model implies that it takes the same amount of time to communicate between any pair of nodes, it corresponds to a completely connected network. Instead of designing algorithms for each specific architecture (for example, a mesh, hypercube, or tree), we can design algorithms with this cost model in mind and port it to any target parallel computer.

This raises the important issue of loss of accuracy (or fidelity) of prediction when the algorithm is ported from our simplified model (which assumes a completely connected network) to an actual machine architecture. If our initial assumption that the t_h term is typically dominated by the t_s or t_w terms is valid, then the loss in accuracy should be minimal.

However, it is important to note that our basic cost model is valid only for uncongested networks. Architectures have varying thresholds for when they get congested; i.e., a linear array has a much lower threshold for congestion than a hypercube. Furthermore, different communication patterns congest a given network to different extents. Consequently, our simplified cost model is valid only as long as the underlying communication pattern does not congest the network.

Example 2.15 Effect of congestion on communication cost

Consider a $\sqrt{p} \times \sqrt{p}$ mesh in which each node is only communicating with its nearest neighbor. Since no links in the network are used for more than one communication, the time for this operation is $t_s + t_w m$, where m is the number of words communicated. This time is consistent with our simplified model.

Consider an alternate scenario in which each node is communicating with a randomly selected node. This randomness implies that there are $p/2$ communications (or $p/4$ bi-directional communications) occurring across any equi-partition of the machine (since the node being communicated with could be in either half with equal probability). From our discussion of bisection width, we know that a 2-D mesh has a bisection width of $\sqrt{p}$. From these two, we can infer that some links would now have to carry at least $\frac{p/4}{\sqrt{p}} = \sqrt{p}/4$ messages, assuming bi-directional communication channels. These messages must be serialized over the link. If each message is of size m, the time for this operation is at least $t_s + t_w m \times \sqrt{p}/4$. This time is not in conformity with our simplified model. ■

The above example illustrates that for a given architecture, some communication patterns can be non-congesting and others may be congesting. This makes the task of modeling communication costs dependent not just on the architecture, but also on the communication pattern. To address this, we introduce the notion of *effective bandwidth*. For communication patterns that do not congest the network, the effective bandwidth is identical to the link bandwidth. However, for communication operations that congest the network, the effective bandwidth is the link bandwidth scaled down by the degree of congestion on the most congested link. This is often difficult to estimate since it is a function of process to node mapping, routing algorithms, and communication schedule. Therefore, we use a lower bound on the message communication time. The associated link bandwidth is scaled down by a factor p/b, where b is the bisection width of the network.

In the rest of this text, we will work with the simplified communication model for message passing with effective per-word time t_w because it allows us to design algorithms in an architecture-independent manner. We will also make specific notes on when a communication operation within an algorithm congests the network and how its impact is factored into parallel runtime. The communication times in the book apply to the general class of k-d meshes. While these times may be realizable on other architectures as well, this is a function of the underlying architecture.

2.5.2 Communication Costs in Shared-Address-Space Machines

The primary goal of 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 for this are as follows:

- 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. This is particularly important in distributed memory shared-address-space architectures because it is difficult to identify local and remote accesses. If the access times for local and remote data items are significantly different, then the cost of communication can vary greatly depending on the data layout.

- Finite cache sizes can result in cache thrashing. Consider a scenario in which a node needs a certain fraction of the total data to compute its results. If this fraction is smaller than locally available cache, the data can be fetched on first access and computed on. However, if the fraction exceeds available cache, then certain portions of this data might get overwritten, and consequently accessed several times. This overhead can cause sharp degradation in program performance as the problem size is increased. To remedy this, the programmer must alter execution schedules (e.g., blocking loops as illustrated in serial matrix multiplication in Problem 2.5) for minimizing working set size. While this problem is common to both serial and multiprocessor platforms, the penalty is much higher in the case of multiprocessors since each miss might now involve coherence operations and interprocessor communication.

- Overheads associated with invalidate and update operations are difficult to quantify. After a data item has been fetched by a processor into cache, it may be subject to a variety of operations at another processor. For example, in an invalidate protocol, the cache line might be invalidated by a write operation at a remote processor. In this case, the next read operation on the data item must pay a remote access latency cost again. Similarly, the overhead associated with an update protocol might vary significantly depending on the number of copies of a data item. The number of concurrent copies of a data item and the schedule of instruction execution are typically beyond the control of the programmer.

- Spatial locality is difficult to model. Since cache lines are generally longer than one word (anywhere from four to 128 words), different words might have different access latencies associated with them even for the first access. Accessing a neighbor of a previously fetched word might be extremely fast, if the cache line has not yet been overwritten. Once again, the programmer has minimal control over this, other than to permute data structures to maximize spatial locality of data reference.

- Prefetching can play a role in reducing the overhead associated with data access. Compilers can advance loads and, if sufficient resources exist, the overhead associated with these loads may be completely masked. Since this is a function of the compiler, the underlying program, and availability of resources (registers/cache), it is very difficult to model accurately.

- False sharing is often an important overhead in many programs. Two words used by (threads executing on) different processor may reside on the same cache line. This may cause coherence actions and communication overheads, even though none of the data might be shared. The programmer must adequately pad data structures used by various processors to minimize false sharing.

- Contention in shared accesses is often a major contributing overhead in shared address space machines. Unfortunately, contention is a function of execution schedule and consequently very difficult to model accurately (independent of the scheduling algorithm). While it is possible to get loose asymptotic estimates by counting the number of shared accesses, such a bound is often not very meaningful.

Any cost model for shared-address-space machines must account for all of these overheads. Building these into a single cost model results in a model that is too cumbersome to design programs for and too specific to individual machines to be generally applicable.

As a first-order model, it is easy to see that accessing a remote word results in a cache line being fetched into the local cache. The time associated with this includes the coherence overheads, network overheads, and memory overheads. The coherence and network overheads are functions of the underlying interconnect (since a coherence operation must be potentially propagated to remote processors and the data item must be fetched). In the absence of knowledge of what coherence operations are associated with a specific access and where the word is coming from, we associate a constant overhead to accessing a cache line of the shared data. For the sake of uniformity with the message-passing model, we refer to this cost as t_s. Because of various latency-hiding protocols, such as prefetching, implemented in modern processor architectures, we assume that a constant cost of t_s is associated with initiating access to a contiguous chunk of *m* words of shared data, even if *m* is greater than the cache line size. We further assume that accessing shared data is costlier than accessing local data (for instance, on a NUMA machine, local data is likely to reside in a local memory module, while data shared by *p* processors will need to be fetched from a nonlocal module for at least *p* - 1 processors). Therefore, we assign a per-word access cost of t_w to shared data.

From the above discussion, it follows that we can use the same expression $t_s + t_w m$ to account for the cost of sharing a single chunk of *m* words between a pair of processors in both shared-memory and message-passing paradigms (Equation 2.4) with the difference that the value of the constant t_s relative to t_w is likely to be much smaller on a shared-memory machine than on a distributed memory machine (t_w is likely to be near zero for a UMA machine). Note that the cost $t_s + t_w m$ assumes read-only access without contention. If multiple processes access the same data, then the cost is multiplied by the number of processes, just as in the message-passing where the process that owns the data will need to send a message to each receiving process. If the access is read-write, then the cost will be incurred again for subsequent access by processors other than the one writing. Once again, there is an equivalence with the message-passing model. If a process modifies the contents of a message that it receives, then it must send it back to processes that subsequently need access to the refreshed data. While this model seems overly simplified in the context of shared-address-space machines, we note that the model provides a good estimate of the cost of sharing an array of *m* words between a pair of processors.

The simplified model presented above accounts primarily for remote data access but does not model a variety of other overheads. Contention for shared data access must be explicitly accounted for by counting the number of accesses to shared data between co-scheduled tasks. The model does not explicitly include many of the other overheads. Since different machines have caches of varying sizes, it is difficult to identify the point at which working set size exceeds the cache size resulting in cache thrashing, in an architecture independent manner. For this reason, effects arising from finite caches are ignored in this cost model. Maximizing spatial locality (cache line effects) is not explicitly included in the cost. False sharing is a function of the instruction schedules as well as data layouts. The cost model assumes that shared data structures are suitably padded and, therefore, does not include false sharing costs. Finally, the cost model does not account for overlapping communication and computation. Other models have been proposed to model overlapped communication. However, designing even simple algorithms for these models is cumbersome. The related issue of multiple concurrent computations (threads) on a single processor is not modeled in the expression. Instead, each processor is assumed to execute a single concurrent unit of computation.

2.6 Routing Mechanisms for Interconnection Networks

Efficient algorithms for routing a message to its destination are critical to the performance of parallel computers. A *routing mechanism* determines the path a message takes through the network to get from the source to the destination node. It takes as input a message's source and destination nodes. It may also use information about the state of the network. It returns one or more paths through the network 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. In a minimal routing scheme, each link brings a message closer to its destination, but the scheme can lead to congestion in parts of the network. A non-minimal routing scheme, in contrast, may route the message along a longer path to avoid network congestion.

Routing mechanisms can also be classified on the basis of how they use information regarding the state of the network. A *deterministic routing* scheme determines a unique path for a message, based on its source and destination. It does not use any information regarding the state of the network. Deterministic schemes may result in uneven use of the communication resources in a network. In contrast, an *adaptive routing* scheme uses information regarding the current state of the network to determine the path of the message. Adaptive routing detects congestion in the network and routes messages around it.

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*.

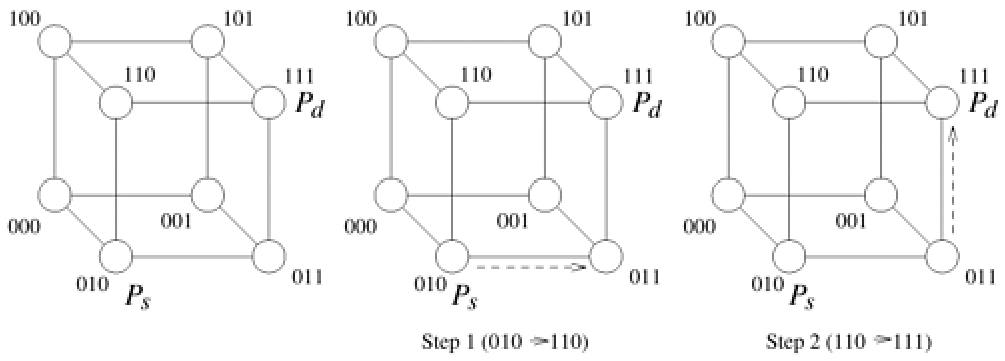
Consider a two-dimensional mesh without wraparound connections. In the XY-routing scheme, a message is sent first along the X dimension until it reaches the column of the destination node and then along the Y dimension until it reaches its destination. Let $P_{Sy,Sx}$ represent the position of the source 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 $|Sx - Dx| + |Sy - Dy|$.

E-cube routing for hypercube-connected networks works similarly. Consider a d-dimensional hypercube of p nodes. Let P_s and P_d be the labels of the source and destination nodes. We know from Section 2.4.3 that the binary representations of these labels are d bits long. Furthermore, 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 illustrates E-cube routing in a three-dimensional hypercube network.

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.



In the rest of this book we assume deterministic and minimal message routing for analyzing parallel algorithms.