- Maximum task size is m.
- It takes a process Δ time to pick up a task.

Compute the best- and worst-case speedups for self-scheduling and chunk-scheduling assuming that tasks are available in batches of l ($l < M$). What are the actual values of the best- and worst-case speedups for the two scheduling methods when $p = 10$, $\Delta = 0.2$, $m = 20$, $M = 100$, and $l = 2$?

PREVIOUS NEXT

Chapter 4. Basic Communication Operations

In most parallel algorithms, processes need to exchange data with other processes. This exchange of data can significantly impact the efficiency of parallel programs by introducing interaction delays during their execution. For instance, recall from Section 2.5 that it takes roughly $t_s + mt_w$ time for a simple exchange of an m-word message between two processes running on different nodes of an interconnection network with cut-through routing. Here t_s is the latency or the startup time for the data transfer and t_w is the per-word transfer time, which is inversely proportional to the available bandwidth between the nodes. Many interactions in practical parallel programs occur in well-defined patterns involving more than two processes. Often either all processes participate together in a single global interaction operation, or subsets of processes participate in interactions local to each subset. These common basic patterns of interprocess interaction or communication are frequently used as building blocks in a variety of parallel algorithms. Proper implementation of these basic communication operations on various parallel architectures is a key to the efficient execution of the parallel algorithms that use them.

In this chapter, we present algorithms to implement some commonly used communication patterns on simple interconnection networks, such as the linear array, two-dimensional mesh, and the hypercube. The choice of these interconnection networks is motivated primarily by pedagogical reasons. For instance, although it is unlikely that large scale parallel computers will be based on the linear array or ring topology, it is important to understand various communication operations in the context of linear arrays because the rows and columns of meshes are linear arrays. Parallel algorithms that perform rowwise or columnwise communication on meshes use linear array algorithms. The algorithms for a number of communication operations on a mesh are simple extensions of the corresponding linear array algorithms to two dimensions. Furthermore, parallel algorithms using regular data structures such as arrays often map naturally onto one- or two-dimensional arrays of processes. This too makes it important to study interprocess interaction on a linear array or mesh interconnection network. The hypercube architecture, on the other hand, is interesting because many algorithms with recursive interaction patterns map naturally onto a hypercube topology. Most of these algorithms may perform equally well on interconnection networks other than the hypercube, but it is simpler to visualize their communication patterns on a hypercube.

The algorithms presented in this chapter in the context of simple network topologies are practical and are highly suitable for modern parallel computers, even though most such computers are unlikely to have an interconnection network that exactly matches one of the networks considered in this chapter. The reason is that on a modern parallel computer, the time to transfer data of a certain size between two nodes is often independent of the relative location of the nodes in the interconnection network. This homogeneity is afforded by a variety of firmware and hardware features such as randomized routing algorithms and cut-through routing, etc. Furthermore, the end user usually does not have explicit control over mapping processes onto physical processors. Therefore, we assume that the transfer of m words of data between *any* pair of nodes in an interconnection network incurs a cost of $t_s + mt_w$. On most architectures, this assumption is reasonably accurate as long as a free link is available between the source and destination nodes for the data to traverse. However, if many pairs of nodes are communicating simultaneously, then the messages may take longer. This can happen if the number of messages passing through a cross-section of the network exceeds the cross-section bandwidth (Section 2.4.4) of the network. In such situations, we need to adjust the value of t_w

to reflect the slowdown due to congestion. As discussed in Section 2.5.1, we refer to the adjusted value of t_w as effective t_w. We will make a note in the text when we come across communication operations that may cause congestion on certain networks.

As discussed in Section 2.5.2, the cost of data-sharing among processors in the shared-address-space paradigm can be modeled using the same expression $t_s + mt_w$, usually with different values of t_s and t_w relative to each other as well as relative to the computation speed of the processors of the parallel computer. Therefore, parallel algorithms requiring one or more of the interaction patterns discussed in this chapter can be assumed to incur costs whose expression is close to one derived in the context of message-passing.

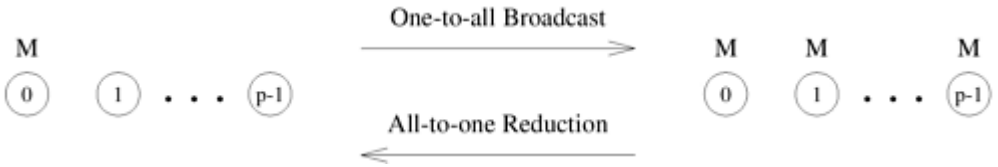
In the following sections we describe various communication operations and derive expressions for their time complexity. We assume that the interconnection network supports cut-through routing (Section 2.5.1) and that the communication time between any pair of nodes is practically independent of of the number of intermediate nodes along the paths between them. We also assume that the communication links are bidirectional; that is, two directly-connected nodes can send messages of size m to each other simultaneously in time $t_s + t_w m$. We assume a single-port communication model, in which a node can send a message on only one of its links at a time. Similarly, it can receive a message on only one link at a time. However, a node can receive a message while sending another message at the same time on the same or a different link.

Many of the operations described here have duals and other related operations that we can perform by using procedures very similar to those for the original operations. The *dual* of a communication operation is the opposite of the original operation and can be performed by reversing the direction and sequence of messages in the original operation. We will mention such operations wherever applicable.

4.1 One-to-All Broadcast and All-to-One Reduction

Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as *one-to-all broadcast* . Initially, only the source process has the data of size *m* that needs to be broadcast. At the termination of the procedure, there are *p* copies of the initial data – one belonging to each process. The dual of one-to-all broadcast is *all-to-one reduction* . In an all-to-one reduction operation, each of the *p* participating processes starts with a buffer *M* containing *m* words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size *m* . Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers – the *i* th word of the accumulated *M* is the sum, product, maximum, or minimum of the *i* th words of each of the original buffers. Figure 4.1 shows one-to-all broadcast and all-to-one reduction among *p* processes.

Figure 4.1. One-to-all broadcast and all-to-one reduction.



One-to-all broadcast and all-to-one reduction are used in several important parallel algorithms including matrix-vector multiplication, Gaussian elimination, shortest paths, and vector inner product. In the following subsections, we consider the implementation of one-to-all broadcast in detail on a variety of interconnection topologies.

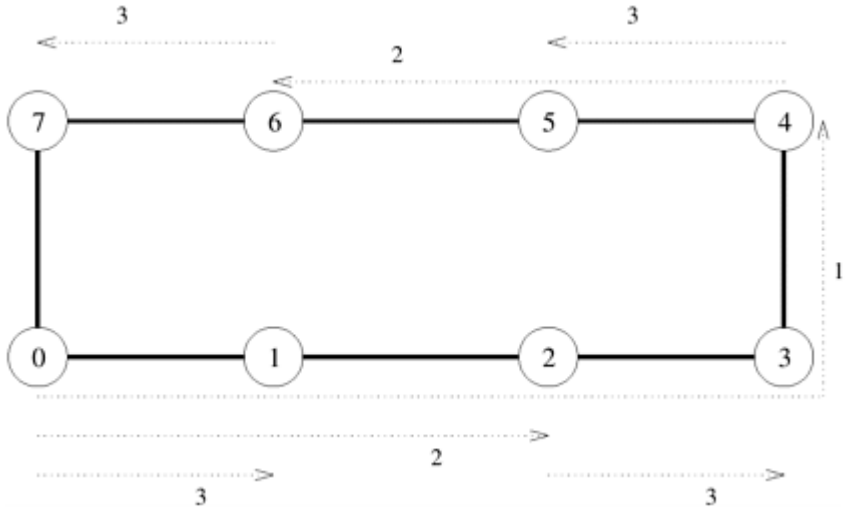
4.1.1 Ring or Linear Array

A naive way to perform one-to-all broadcast is to sequentially send *p* - 1 messages from the source to the other *p* - 1 processes. However, this is inefficient because the source process becomes a bottleneck. Moreover, the communication network is underutilized because only the connection between a single pair of nodes is used at a time. A better broadcast algorithm can be devised using a technique commonly known as *recursive doubling* . The source process first sends the message to another process. Now both these processes can simultaneously send the message to two other processes that are still waiting for the message. By continuing this procedure until all the processes have received the data, the message can be broadcast in log *p* steps.

The steps in a one-to-all broadcast on an eight-node linear array or ring are shown in Figure 4.2 . The nodes are labeled from 0 to 7. Each message transmission step is shown by a numbered, dotted arrow from the source of the message to its destination. Arrows indicating messages sent during the same time step have the same number.

Figure 4.2. One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during

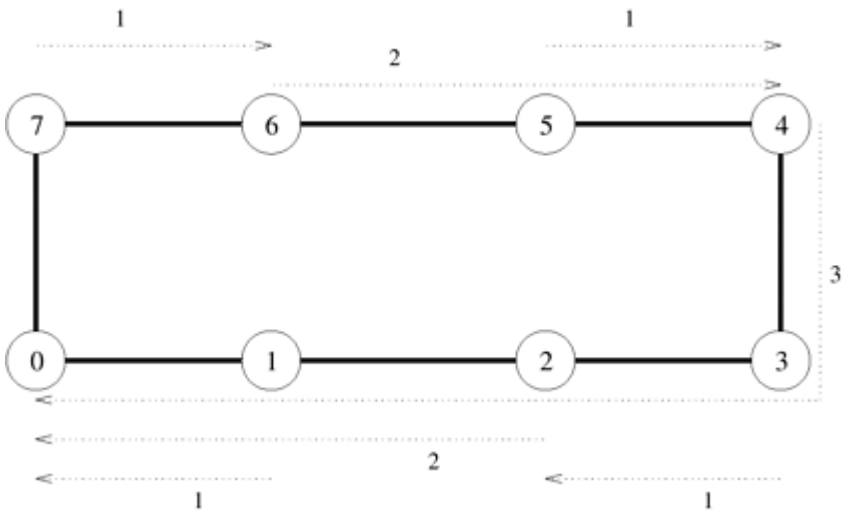
which the message is transferred.



Note that on a linear array, the destination node to which the message is sent in each step must be carefully chosen. In Figure 4.2 , the message is first sent to the farthest node (4) from the source (0). In the second step, the distance between the sending and receiving nodes is halved, and so on. The message recipients are selected in this manner at each step to avoid congestion on the network. For example, if node 0 sent the message to node 1 in the first step and then nodes 0 and 1 attempted to send messages to nodes 2 and 3, respectively, in the second step, the link between nodes 1 and 2 would be congested as it would be a part of the shortest route for both the messages in the second step.

Reduction on a linear array can be performed by simply reversing the direction and the sequence of communication, as shown in Figure 4.3 . In the first step, each odd numbered node sends its buffer to the even numbered node just before itself, where the contents of the two buffers are combined into one. After the first step, there are four buffers left to be reduced on nodes 0, 2, 4, and 6, respectively. In the second step, the contents of the buffers on nodes 0 and 2 are accumulated on node 0 and those on nodes 6 and 4 are accumulated on node 4. Finally, node 4 sends its buffer to node 0, which computes the final result of the reduction.

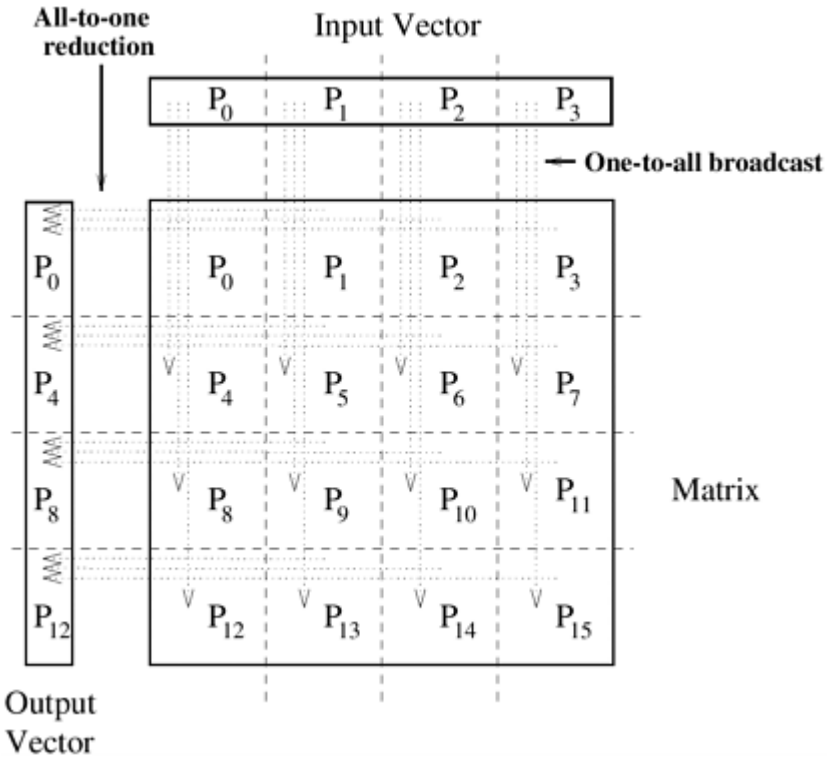
Figure 4.3. Reduction on an eight-node ring with node 0 as the destination of the reduction.



Example 4.1 Matrix-vector multiplication

Consider the problem of multiplying an n x n matrix A with an n x 1 vector x on an n x n mesh of nodes to yield an n x 1 result vector y . Algorithm 8.1 shows a serial algorithm for this problem. Figure 4.4 shows one possible mapping of the matrix and the vectors in which each element of the matrix belongs to a different process, and the vector is distributed among the processes in the topmost row of the mesh and the result vector is generated on the leftmost column of processes.

Figure 4.4. One-to-all broadcast and all-to-one reduction in the multiplication of a 4 x 4 matrix with a 4 x 1 vector.



Since all the rows of the matrix must be multiplied with the vector, each process needs the element of the vector residing in the topmost process of its column. Hence, before computing the matrix-vector product, each column of nodes performs a one-to-all broadcast of the vector elements with the topmost process of the column as the source. This is done by treating each column of the n x n mesh as an n -node linear array, and simultaneously applying the linear array broadcast procedure described previously to all columns.

After the broadcast, each process multiplies its matrix element with the result of the broadcast. Now, each row of processes needs to add its result to generate the corresponding element of the product vector. This is accomplished by performing all-to-one reduction on each row of the process mesh with the first process of each row as the destination of the reduction operation.

For example, P_9 will receive x [1] from P_1 as a result of the broadcast, will multiply it with A [2, 1] and will participate in an all-to-one reduction with P_8 , P_{10} , and P_{11} to accumulate y [2] on P_8 . ■