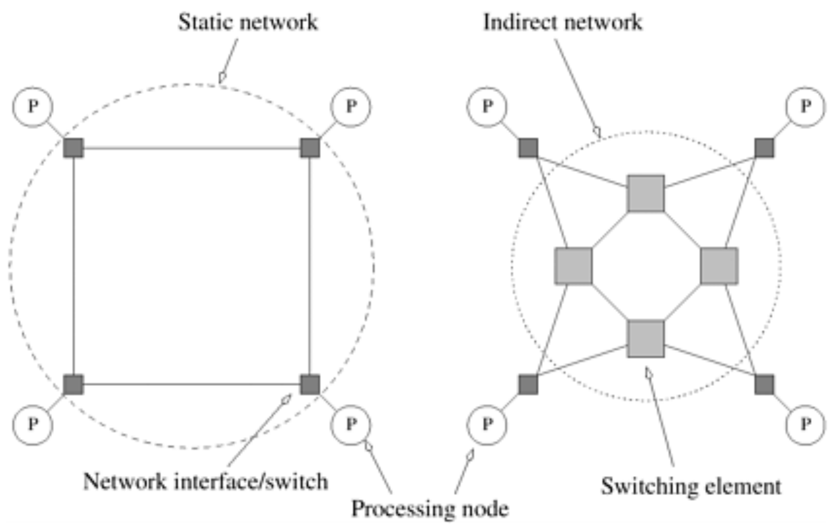


~~A single switch in an interconnection network consists of a set of input ports and a set of output ports. Switches provide a range of functionality. The minimal functionality provided by a switch is a mapping from the input to the output ports. The total number of ports on a switch is also called the ***degree*** of the switch. Switches may also provide support for internal buffering (when the requested output port is busy), routing (to alleviate congestion on the network), and multicast (same output on multiple ports). The mapping from input to output ports can be provided using a variety of mechanisms based on physical crossbars, multi-ported memories, multiplexor-demultiplexors, and multiplexed buses. The cost of a switch is influenced by the cost of the mapping hardware, the peripheral hardware and packaging costs. The mapping hardware typically grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.~~

~~The connectivity between the nodes and the network is provided by a network interface. The network interface has input and output ports that pipe data into and out of the network. It typically has the responsibility of packetizing data, computing routing information, buffering incoming and outgoing data for matching speeds of network and processing elements, and error checking. The position of the interface between the processing element and the network is also important. While conventional network interfaces hang off the I/O buses, interfaces in tightly coupled parallel machines hang off the memory bus. Since I/O buses are typically slower than memory buses, the latter can support higher bandwidth.~~

2.4.3 Network Topologies

A wide variety of network topologies have been used in interconnection networks. These topologies try to trade off cost and scalability with performance. While pure topologies have attractive mathematical properties, in practice interconnection networks tend to be combinations or modifications of the pure topologies discussed in this section.

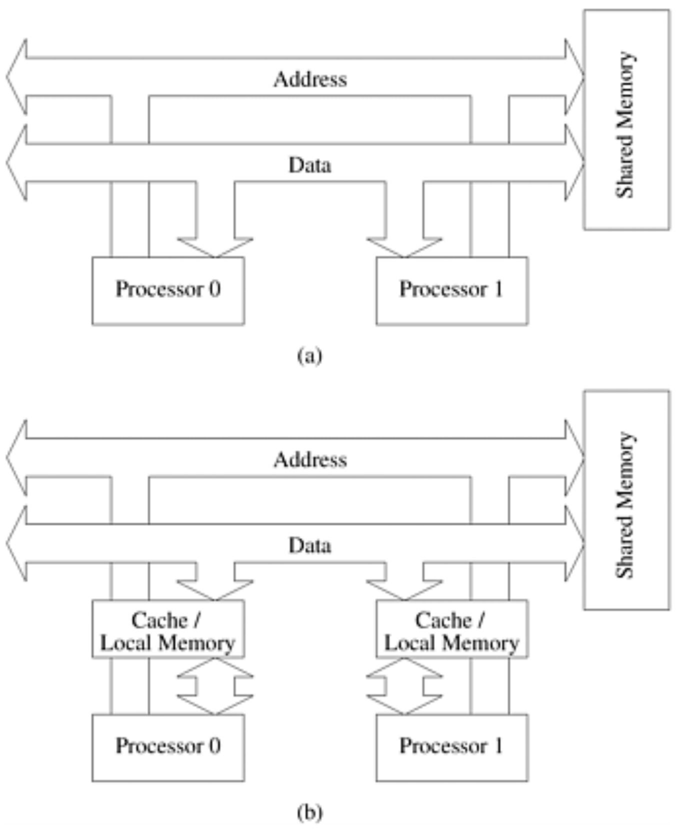
Bus-Based Networks

A bus-based network is perhaps the simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network scales linearly as the number of nodes, *p*. This cost is typically associated with bus interfaces. Furthermore, the distance between any two nodes in the network is constant ($O(1)$). Buses are also ideal for broadcasting information among nodes. Since the transmission medium is shared, there is little overhead associated with broadcast compared to point-to-point message transfer. However, the bounded bandwidth of a bus places limitations on the overall performance of the network as the number of nodes increases. Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

The demands on bus bandwidth can be reduced by making use of the property that in typical programs, a majority of the data accessed is local to the node. For such programs, it is possible to provide a cache for each node. Private data is cached at the node and only remote data is accessed through the bus.

Example 2.12 Reducing shared-bus bandwidth using caches Figure 2.7(a) illustrates *p* processors sharing a bus to the memory. Assuming that each processor accesses *k* data items, and each data access takes time t_{cycle}, the execution time is lower bounded by t_{cycle} x *kp* seconds. Now consider the hardware organization of Figure 2.7(b). Let us assume that 50% of the memory accesses (0.5*k*) are made to local data. This local data resides in the private memory of the processor. We assume that access time to the private memory is identical to the global memory, i.e., t_{cycle}. In this case, the total execution time is lower bounded by 0.5 x t_{cycle} x *k* + 0.5 x t_{cycle} x *kp*. Here, the first term results from accesses to local data and the second term from access to shared data. It is easy to see that as *p* becomes large, the organization of Figure 2.7(b) results in a lower bound that approaches 0.5 x t_{cycle} x *kp*. This time is a 50% improvement in lower bound on execution time compared to the organization of Figure 2.7(a). ■

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

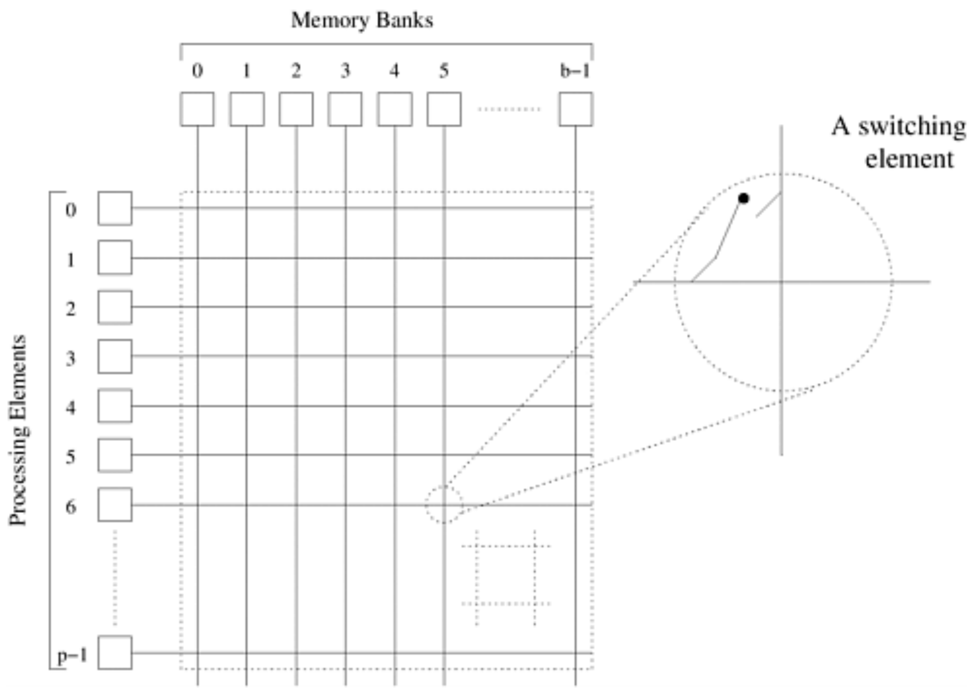


In practice, shared and private data is handled in a more sophisticated manner. This is briefly addressed with cache coherence issues in Section 2.4.6.

Crossbar Networks

A simple way to connect *p* processors to *b* memory banks is to use a crossbar network. A crossbar network employs a grid of switches or switching nodes as shown in Figure 2.8. The crossbar network is a non-blocking network in the sense that the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

Figure 2.8. A completely non-blocking crossbar network connecting *p* processors to *b* memory banks.



The total number of switching nodes required to implement such a network is Q(*pb*). It is reasonable to assume that the number of memory banks *b* is at least *p*; otherwise, at any given time, there will be some processing nodes that will be unable to access any memory banks. Therefore, as the value of *p* is increased, the complexity (component count) of the switching network grows as W(p^2). (See the Appendix for an explanation of the W notation.) As the number of processing nodes becomes large, this switch complexity is difficult to realize at high data rates. Consequently, crossbar networks are not very scalable in terms of cost.

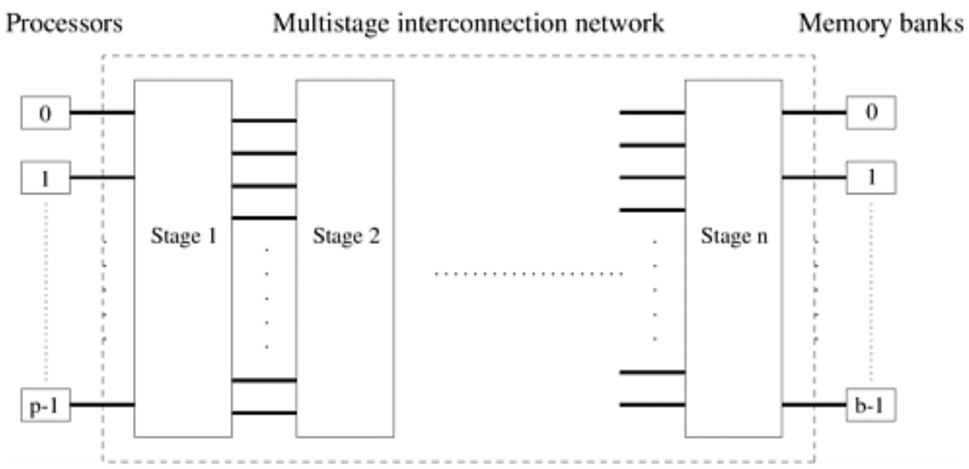
Multistage Networks

The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called ***multistage interconnection networks*** lies between these two extremes. It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

The general schematic of a multistage network consisting of *p* processing nodes and *b* memory banks is shown in Figure 2.9. A commonly used multistage connection network is the ***omega network***. This network consists of log *p* stages, where *p* is the number of inputs (processing nodes) and also the number of outputs (memory banks). Each stage of the omega network consists of an interconnection pattern that connects *p* inputs and *p* outputs; a link exists between input *i* and output *j* if the following is true: **Equation 2.1**

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

Figure 2.9. The schematic of a typical multistage interconnection network.



Equation 2.1 represents a left-rotation operation on the binary representation of *i* to obtain *j*. This interconnection pattern is called a ***perfect shuffle***. Figure 2.10 shows a perfect shuffle interconnection pattern for eight inputs and outputs. At each stage of an omega network, a perfect shuffle interconnection pattern feeds into a set of *p*/2 switches or switching nodes. Each switch is in one of two connection modes. In one mode, the inputs are sent straight through to the outputs, as shown in Figure 2.11(a). This is called the ***pass-through*** connection. In the other mode, the inputs to the switching node are crossed over and then sent out, as shown in Figure 2.11(b). This is called the ***cross-over*** connection.

Figure 2.10. A perfect shuffle interconnection for eight inputs and outputs.

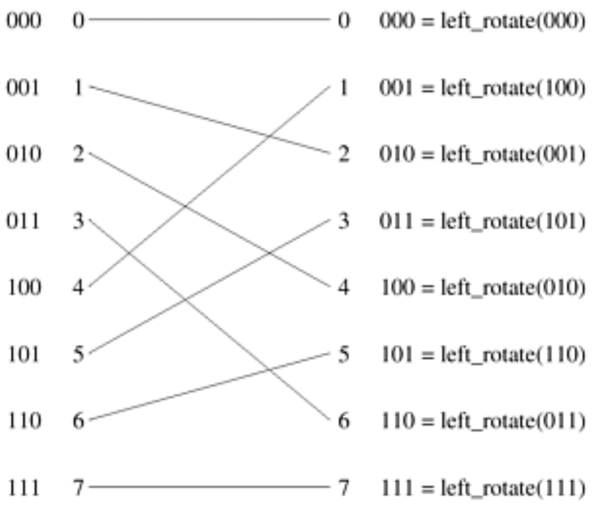
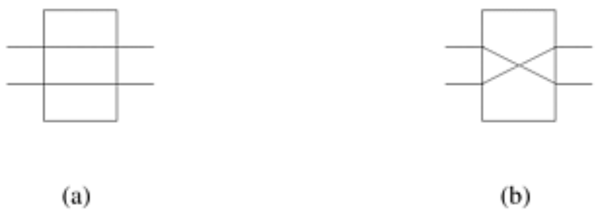


Figure 2.11. Two switching configurations of the 2 x 2 switch: (a) Pass-through; (b) Cross-over.



An omega network has *p*/2 x log *p* switching nodes, and the cost of such a network grows as Q(*p* log *p*). Note that this cost is less than the Q(p^2) cost of a complete crossbar network. Figure 2.12 shows an omega network for eight processors (denoted by the binary numbers on the left) and eight memory banks (denoted by the binary numbers on the right). Routing data in an omega network is accomplished using a simple scheme. Let *s* be the binary representation of a processor that needs to write some data into memory bank *t*. The data traverses the link to the first switching node. If the most significant bits of *s* and *t* are the same, then the data is routed in pass-through mode by the switch. If these bits are different, then the data is routed through in crossover mode. This scheme is repeated at the next switching stage using the next most significant bit. Traversing log *p* stages uses all log *p* bits in the binary representations of *s* and *t*.

Figure 2.12. A complete omega network connecting eight inputs and eight outputs.

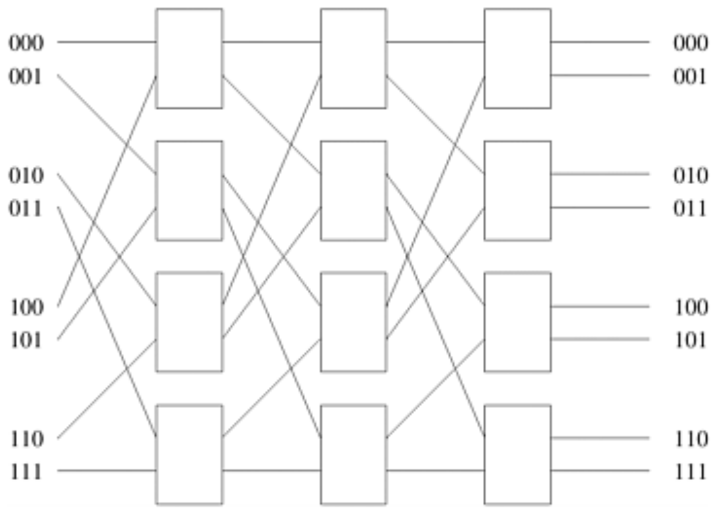
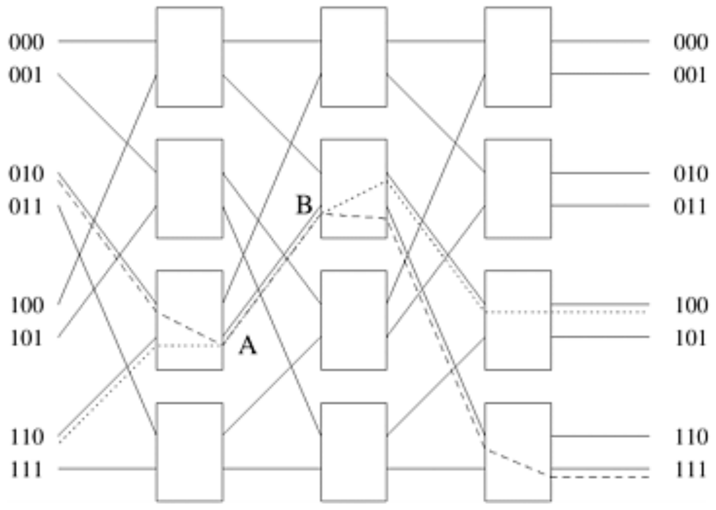


Figure 2.13 shows data routing over an omega network from processor two (010) to memory bank seven (111) and from processor six (110) to memory bank four (100). This figure also illustrates an important property of this network. When processor two (010) is communicating with memory bank seven (111), it blocks the path from processor six (110) to memory bank four (100). Communication link AB is used by both communication paths. Thus, in an omega network, access to a memory bank by

a processor may disallow access to another memory bank by another processor. Networks with this property are referred to as ***blocking networks***.

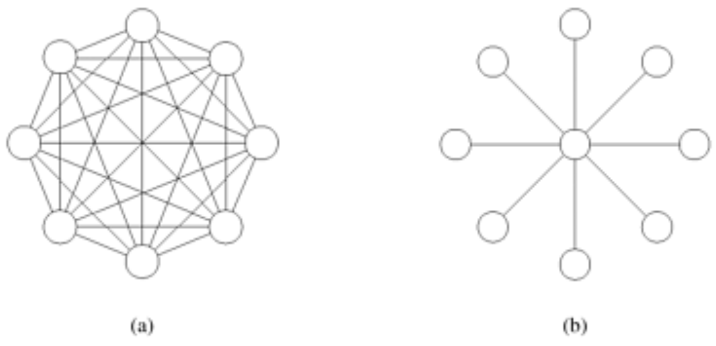
Figure 2.13. An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.



Completely-Connected Network

In a ***completely-connected network***, each node has a direct communication link to every other node in the network. Figure 2.14(a) illustrates a completely-connected network of eight nodes. This network is ideal in the sense that a node can send a message to another node in a single step, since a communication link exists between them. Completely-connected networks are the static counterparts of crossbar switching networks, since in both networks, the communication between any input/output pair does not block communication between any other pair.

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



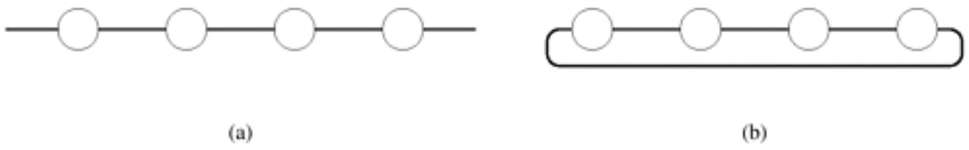
Star-Connected Network

In a ***star-connected network***, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor. Figure 2.14(b) shows a star-connected network of nine processors. The star-connected network is similar to bus-based networks. Communication between any pair of processors is routed through the central processor, just as the shared bus forms the medium for all communication in a bus-based network. The central processor is the bottleneck in the star topology.

Linear Arrays, Meshes, and *k-d* Meshes

Due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers. A family of such networks spans the space of linear arrays and hypercubes. A linear array is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right. A simple extension of the linear array (Figure 2.15(a)) is the ring or a 1-D torus (Figure 2.15(b)). The ring has a wraparound connection between the extremities of the linear array. In this case, each node has two neighbors.

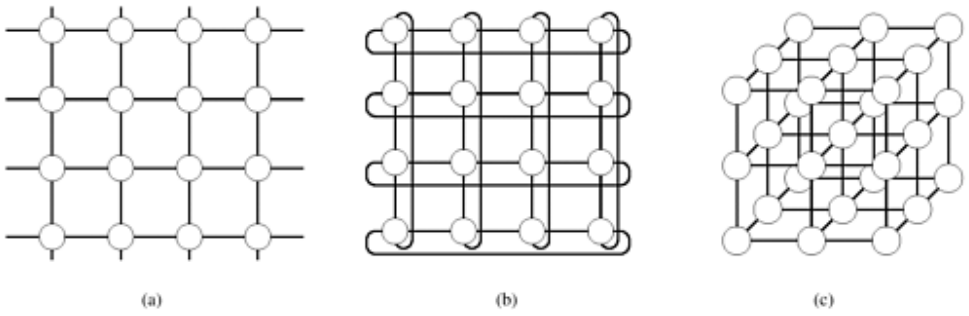
Figure 2.15. Linear arrays: (a) with no wraparound links; (b) with wraparound link.



A two-dimensional mesh illustrated in Figure 2.16(a) is an extension of the linear array to two-dimensions. Each dimension has $\sqrt{p}$ nodes with a node identified by a two-tuple (*i*, *j*). Every node (except those on the periphery) is connected to four other nodes whose indices differ in any dimension by one. A 2-D mesh has the property that it can be laid out in 2-D space, making it attractive from a wiring standpoint. Furthermore, a variety of regularly structured computations map very naturally to a 2-D mesh. For this reason, 2-D meshes were often used as interconnects in parallel machines. Two dimensional meshes can be augmented with wraparound links to form two dimensional tori illustrated in Figure 2.16(b). The three-dimensional cube is a generalization of the 2-D mesh to three dimensions, as illustrated in Figure 2.16(c). Each node element in a 3-D cube, with the exception of those on the periphery, is connected to six other nodes, two along each of the three dimensions. A variety of physical simulations commonly executed on parallel computers (for example, 3-D weather modeling, structural modeling, etc.) can be mapped naturally to 3-D network topologies. For this reason, 3-D cubes are used commonly in interconnection networks for parallel computers (for example, in the Cray T3E).

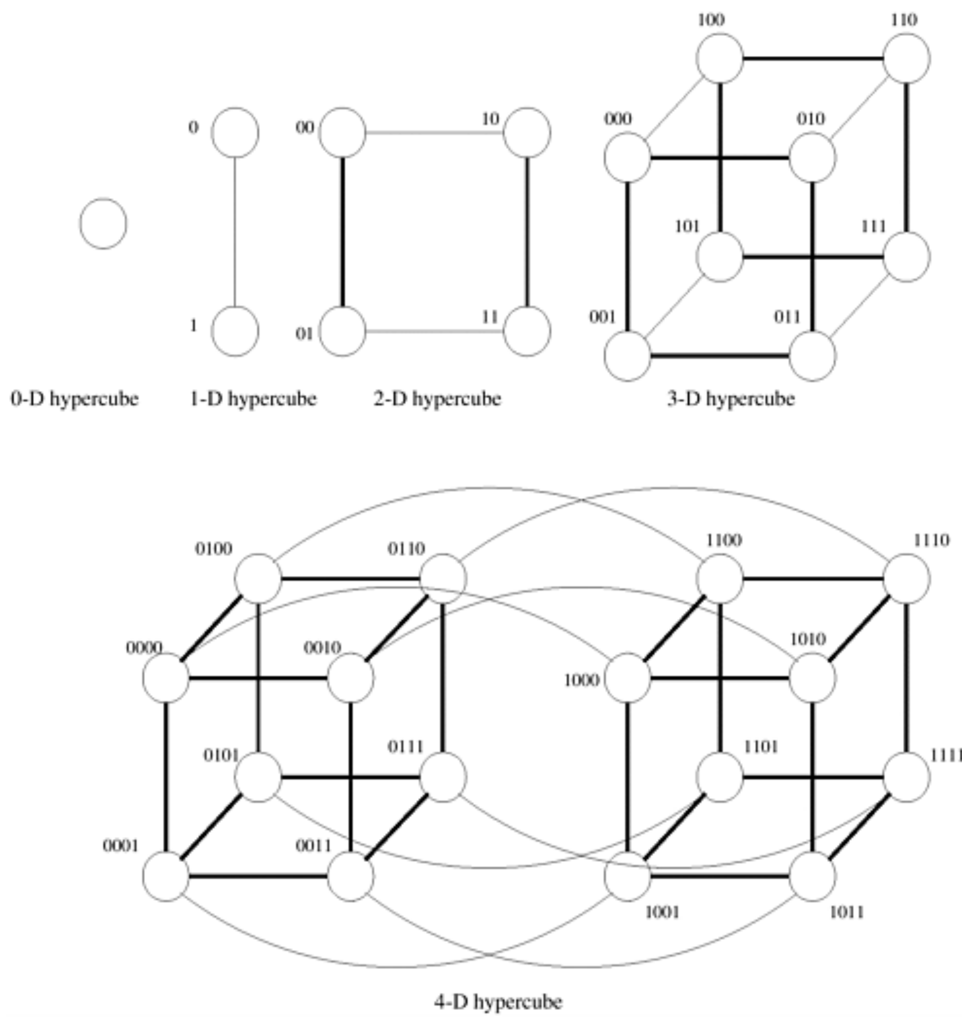
Figure 2.16. Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus);

and (c) a 3-D mesh with no wraparound.



The general class of *k*-*d* meshes refers to the class of topologies consisting of *d* dimensions with *k* nodes along each dimension. Just as a linear array forms one extreme of the *k*-*d* mesh family, the other extreme is formed by an interesting topology called the hypercube. The hypercube topology has two nodes along each dimension and log *p* dimensions. The construction of a hypercube is illustrated in Figure 2.17. A zero-dimensional hypercube consists of 2^0, i.e., one node. A one-dimensional hypercube is constructed from two zero-dimensional hypercubes by connecting them. A two-dimensional hypercube of four nodes is constructed from two one-dimensional hypercubes by connecting corresponding nodes. In general a *d*-dimensional hypercube is constructed by connecting corresponding nodes of two (*d* - 1) dimensional hypercubes. Figure 2.17 illustrates this for up to 16 nodes in a 4-D hypercube.

Figure 2.17. Construction of hypercubes from hypercubes of lower dimension.

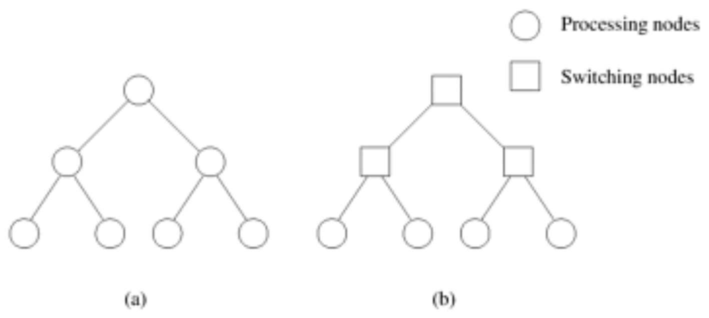


It is useful to derive a numbering scheme for nodes in a hypercube. A simple numbering scheme can be derived from the construction of a hypercube. As illustrated in Figure 2.17, if we have a numbering of two subcubes of *p*/2 nodes, we can derive a numbering scheme for the cube of *p* nodes by prefixing the labels of one of the subcubes with a "0" and the labels of the other subcube with a "1". This numbering scheme has the useful property that the minimum distance between two nodes is given by the number of bits that are different in the two labels. For example, nodes labeled 0110 and 0101 are two links apart, since they differ at two bit positions. This property is useful for deriving a number of parallel algorithms for the hypercube architecture.

Tree-Based Networks

A ***tree network*** is one in which there is only one path between any pair of nodes. Both linear arrays and star-connected networks are special cases of tree networks. Figure 2.18 shows networks based on complete binary trees. Static tree networks have a processing element at each node of the tree (Figure 2.18(a)). Tree networks also have a dynamic counterpart. In a dynamic tree network, nodes at intermediate levels are switching nodes and the leaf nodes are processing elements (Figure 2.18(b)).

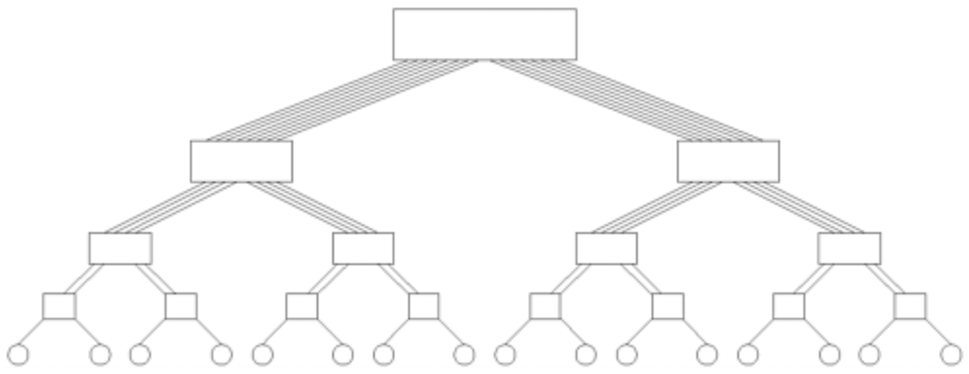
Figure 2.18. Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.



To route a message in a tree, the source node sends the message up the tree until it reaches the node at the root of the smallest subtree containing both the source and destination nodes. Then the message is routed down the tree towards the destination node.

Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many nodes in the left subtree of a node communicate with nodes in the right subtree, the root node must handle all the messages. This problem can be alleviated in dynamic tree networks by increasing the number of communication links and switching nodes closer to the root. This network, also called a ***fat tree***, is illustrated in Figure 2.19.

Figure 2.19. A fat tree network of 16 processing nodes.



not in syllabus

2.4.4 Evaluating Static Interconnection Networks

We now discuss various criteria used to characterize the cost and performance of static interconnection networks. We use these criteria to evaluate static networks introduced in the previous subsection.

6.1 Principles of Message-Passing Programming

There are two key attributes that characterize the message-passing programming paradigm. The first is that it assumes a partitioned address space and the second is that it supports only explicit parallelization.

The logical view of a machine supporting the message-passing paradigm consists of *p* processes, each with its own exclusive address space. Instances of such a view come naturally from clustered workstations and non-shared address space multicomputers. There are two immediate implications of a partitioned address space. First, each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming, but encourages locality of access that is critical for achieving high performance on non-UMA architecture, since a processor can access its local data much faster than non-local data on such architectures. The second implication is that all interactions (read-only or read/write) require cooperation of two processes � the process that has the data and the process that wants to access the data. This requirement for cooperation adds a great deal of complexity for a number of reasons. The process that has the data must participate in the interaction even if it has no logical connection to the events at the requesting process. In certain circumstances, this requirement leads to unnatural programs. In particular, for dynamic and/or unstructured interactions the complexity of the code written for this type of paradigm can be very high for this reason. However, a primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions, and is more likely to think about algorithms (and mappings) that minimize interactions. Another major advantage of this type of programming paradigm is that it can be efficiently implemented on a wide variety of architectures.

The message-passing programming paradigm requires that the parallelism is coded explicitly by the programmer. That is, the programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency. As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding. However, on the other hand, properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.

Structure of Message-Passing Programs Message-passing programs are often written using the ***asynchronous*** or ***loosely synchronous*** paradigms. In the asynchronous paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about, and can have non-deterministic behavior due to race conditions. Loosely synchronous programs are a good compromise between these two extremes. In such programs, tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Since the interaction happens synchronously, it is still quite easy to reason about the program. Many of the known parallel algorithms can be naturally implemented using loosely synchronous programs.

In its most general form, the message-passing paradigm supports execution of a different program on each of the *p* processes. This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable. For this reason, most message-passing programs are written using the

single program multiple data (SPMD) approach. In SPMD programs the code executed by different processes is identical except for a small number of processes (e.g., the "root" process). This does not mean that the processes work in lock-step. In an extreme case, even in an SPMD program, each process could execute a different code (the program contains a large case statement with code for each process). But except for this degenerate case, most processes execute the same code. SPMD programs can be loosely synchronous or completely asynchronous.

6.2 The Building Blocks: Send and Receive Operations

Since interactions are accomplished by sending and receiving messages, the basic operations in the message-passing programming paradigm are `send` and `receive`. In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

The `sendbuf` points to a buffer that stores the data to be sent, `recvbuf` points to a buffer that stores the data to be received, `nelems` is the number of data units to be sent and received, `dest` is the identifier of the process that receives the data, and `source` is the identifier of the process that sends the data.

However, to stop at this point would be grossly simplifying the programming and performance ramifications of how these functions are implemented. To motivate the need for further investigation, let us start with a simple example of a process sending a piece of data to another process as illustrated in the following code-fragment:

```
        P0                                  P1

        a = 100;                            receive(&a, 1, 0)
        send(&a, 1, 1);                     printf("%d\n", a);
        a=0;
```

In this simple example, process P0 sends a message to process P1 which receives and prints the message. The important thing to note is that process P0 changes the value of `a` to 0 immediately following the `send`. The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0. That is, the value of `a` at the time of the send operation must be the value that is received by process `P1`.

It may seem that it is quite straightforward to ensure the semantics of the send and receive operations. However, based on how the send and receive operations are implemented this may not be the case. Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware. Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention. Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) without CPU support (once they have been programmed). As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process `P1` might receive the value 0 in `a` instead of 100!

While this is undesirable, there are in fact reasons for supporting such send operations for performance reasons. In the rest of this section, we will discuss send and receive operations in the context of such a hardware environment, and motivate various implementation details and message-passing protocols that help in ensuring the semantics of the send and receive operations.

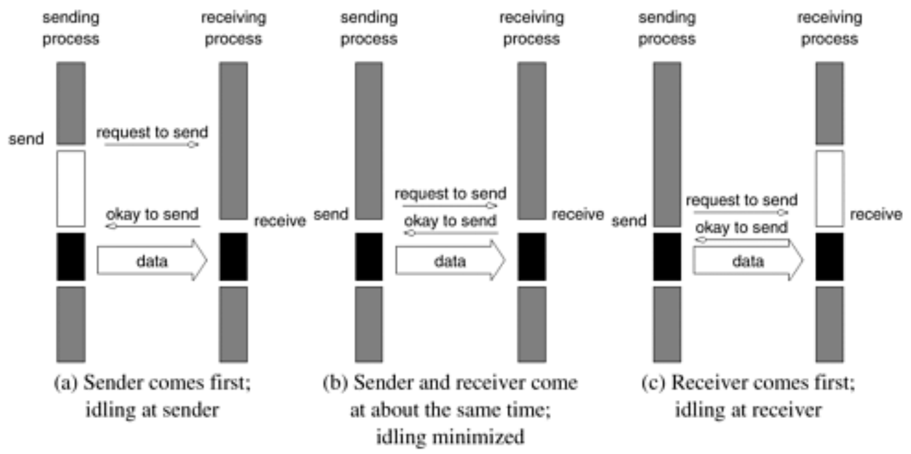
6.2.1 Blocking Message Passing Operations

A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so. Note that this is not the same as saying that the send operation returns only after the receiver has received the data. It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently. There are two mechanisms by which this can be achieved.

Blocking Non-Buffered Send/Receive

In the first case, the send operation does not return until the matching receive has been encountered at the receiving process. When this happens, the message is sent and the send operation returns upon completion of the communication operation. Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process. When the receiving process encounters the target receive, it responds to the request. The sending process upon receiving this response initiates a transfer operation. The operation is illustrated in Figure 6.1. Since there are no buffers used at either sending or receiving ends, this is also referred to as a ***non-buffered blocking operation***.

Figure 6.1. Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



Idling Overheads in Blocking Non-Buffered Operations In Figure 6.1, we illustrate three scenarios in which the send is reached before the receive is posted, the send and receive are posted around the same time, and the receive is posted before the send is reached. In cases (a) and (c), we notice that there is considerable idling at the sending and receiving process. It is also clear from the figures that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time. However, in an asynchronous environment, this may be impossible to predict. This idling overhead is one of the major drawbacks of this protocol.

Deadlocks in Blocking Non-Buffered Operations Consider the following simple exchange of messages that can lead to a deadlock:

```
           P0                                  P1

```

```
          send(&a, 1, 1);                     send(&a, 1, 0);
          receive(&b, 1, 1);                  receive(&b, 1, 0);
```

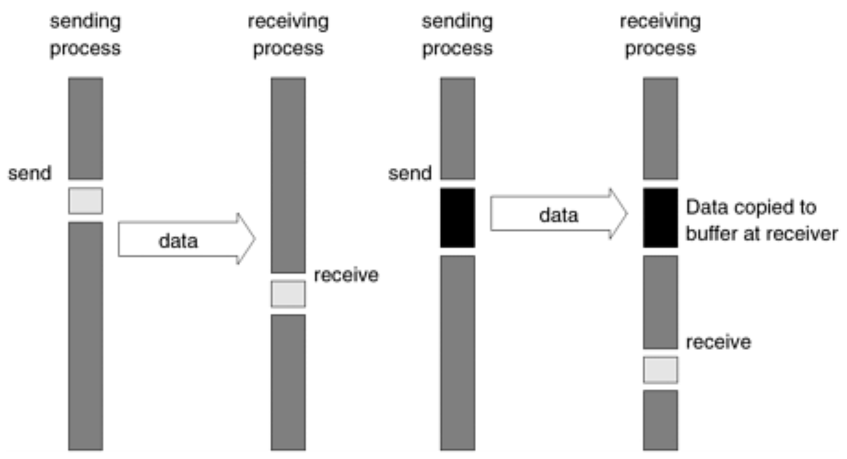
The code fragment makes the values of a available to both processes P0 and P1. However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.

As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined. In the above example, this can be corrected by replacing the operation sequence of one of the processes by a receive and a send as opposed to the other way around. This often makes the code more cumbersome and buggy.

Blocking Buffered Send/Receive

A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. We start with a simple case in which the sender has a buffer pre-allocated for communicating messages. On encountering a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The sender process can now continue with the program knowing that any changes to the data will not impact program semantics. The actual communication can be accomplished in many ways depending on the available hardware resources. If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer. Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well. When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location. This operation is illustrated in Figure 6.2(a).

Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.



In the protocol illustrated above, buffers are used at both sender and receiver and communication is handled by dedicated hardware. Sometimes machines do not have such communication hardware. In this case, some of the overhead can be saved by buffering only on one side. For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end. When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location. This protocol is illustrated in Figure 6.2(b). It is not difficult to conceive a protocol in which the buffering is done only at the sender and the receiver initiates a transfer by interrupting the sender.

It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads. In general, if the parallel program is highly synchronous (i.e., sends and receives are posted around the same time), non-buffered sends may perform better than buffered sends. However, in general applications, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.

Example 6.1 Impact of finite buffers in message passing

Consider the following code fragment:

```
P0                                     P1

for (i = 0; i < 1000; i++) {           for (i = 0; i < 1000; i++) {
  produce_data(&a);                      receive(&a, 1, 0);
  send(&a, 1, 1);                        consume_data(&a);
}                                      }
```

In this code fragment, process P0 produces 1000 data items and process P1 consumes them. However, if process P1 was slow getting to this loop, process P0 might have sent all of its data. If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space. This can often lead to unforeseen overheads and performance degradation. In general, it is a good idea to write programs that have bounded buffer requirements. ■

Deadlocks in Buffered Send and Receive Operations While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks. This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

```
P0                                    P1

receive(&a, 1, 1);                    receive(&a, 1, 0);
send(&b, 1, 1);                       send(&b, 1, 0);
```

Once again, such circular waits have to be broken. However, deadlocks are caused only by waits on receive operations in this case.

6.2.2 Non-Blocking Message Passing Operations

In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered). Often, it is possible to require the programmer to ensure semantic correctness and provide a fast send/receive operation that incurs little overhead. This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so. Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation. Non-blocking operations are generally accompanied by a `check-status` operation, which indicates whether the semantics of a previously initiated transfer may be violated or not. Upon return from a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation. Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.

As illustrated in Figure 6.3, non-blocking operations can themselves be buffered or non-buffered. In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work. At some point in the future, when the corresponding receive is posted, the communication operation is initiated. When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data. This transfer is indicated in Figure 6.4(a).

Figure 6.3. Space of possible protocols for send and receive operations.

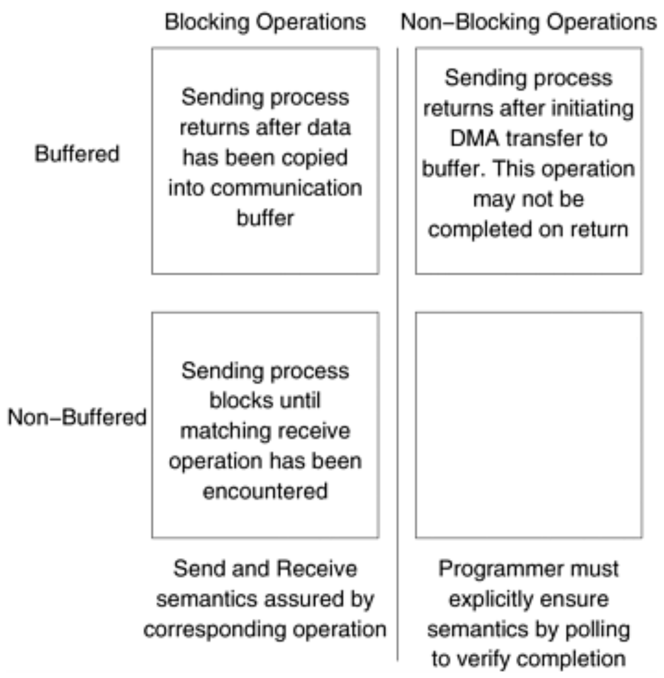
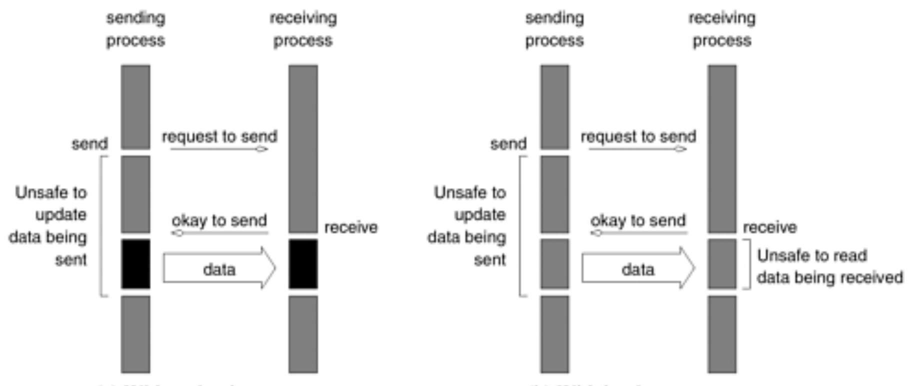

Blocking Operations
Non–Blocking Operations
Buffered
Non–Buffered
Sending process returns after data has been copied into communication buffer
Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Sending process blocks until matching receive operation has been encountered
Send and Receive semantics assured by corresponding operation
Programmer must explicitly ensure semantics by polling to verify completion

Figure 6.4. Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.



(a) Without hardware support (b) With hardware support

Comparing Figures 6.4(a) and 6.1(a), it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation, provided it does not update the data being sent. This alleviates the major bottleneck associated with the former at the expense of some program restructuring. The benefits of non-blocking operations are further enhanced by the presence of dedicated communication hardware. This is illustrated in Figure 6.4(b). In this case, the communication overhead can be almost entirely masked by non-blocking operations. In this case, however, the data being received is unsafe for the duration of the receive operation.

Non-blocking operations can also be used with a buffered protocol. In this case, the sender initiates a DMA operation and returns immediately. The data becomes safe the moment the DMA operation has been completed. At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location. Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations. Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead. One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.

6.3 MPI: the Message Passing Interface

Many early generation commercial parallel computers were based on the message-passing architecture due to its lower cost relative to shared-address-space architectures. Since message-passing is the natural programming paradigm for these machines, this resulted in the development of many different message-passing libraries. In fact, message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library, that performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors. Many of the differences between the various vendor-specific message-passing libraries were only syntactic; however, often enough there were some serious semantic differences that required significant re-engineering to port a message-passing program from one library to another.

The message-passing interface, or MPI as it is commonly known, was created to essentially solve this problem. MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran. The MPI standard defines both the syntax as well as the semantics of a core set of library routines that are very useful in writing message-passing programs. MPI was developed by a group of researchers from academia and industry, and has enjoyed wide support by almost all the hardware vendors. Vendor implementations of MPI are available on almost all commercial parallel computers.

The MPI library contains over 125 routines, but the number of key concepts is much smaller. In fact, it is possible to write fully-functional message-passing programs by using only the six routines shown in Table 6.1. These routines are used to initialize and terminate the MPI library, to get information about the parallel computing environment, and to send and receive messages.

In this section we describe these routines as well as some basic concepts that are essential in writing correct and efficient message-passing programs using MPI.

Table 6.1. The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

6.3.1 Starting and Terminating the MPI Library

`MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment. Calling `MPI_Init` more than once during the execution of a program will lead to an error. `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment. No MPI calls may be performed after `MPI_Finalize` has been called, not even `MPI_Init`. Both `MPI_Init` and `MPI_Finalize` must be called by all the processes, otherwise MPI's behavior will be undefined. The exact calling sequences of these two routines for C are as follows: int MPI_Init(int *argc, char ***argv) int MPI_Finalize()

The arguments argc and argv of MPI_Init are the command-line arguments of the C program. An MPI implementation is expected to remove from the argv array any command-line arguments that should be processed by the implementation before returning back to the program, and to decrement argc accordingly. Thus, command-line processing should be performed only after MPI_Init has been called. Upon successful execution, MPI_Init and MPI_Finalize return MPI_SUCCESS; otherwise they return an implementation-defined error code.

The bindings and calling sequences of these two functions are illustrative of the naming practices and argument conventions followed by MPI. All MPI routines, data-types, and constants are prefixed by "MPI_". The return code for successful completion is MPI_SUCCESS. This and other MPI constants and data-structures are defined for C in the file "mpi.h". This header file must be included in each MPI program.

6.3.2 Communicators

A key concept used throughout MPI is that of the *communication domain*. A communication domain is a set of processes that are allowed to communicate with each other. Information about communication domains is stored in variables of type MPI_Comm,that are called ***communicators***. These communicators are used as arguments to all message transfer MPI routines and they uniquely identify the processes participating in the message transfer operation. Note that each process can belong to many different (possibly overlapping) communication domains.

The communicator is used to define a set of processes that can communicate with each other. This set of processes form a ***communication domain***. In general, all the processes may need to communicate with each other. For this reason, MPI defines a default communicator called MPI_COMM_WORLD which includes all the processes involved in the parallel execution. However, in many cases we want to perform communication only within (possibly overlapping) groups of processes. By using a different communicator for each such group, we can ensure that no messages will ever interfere with messages destined to any other group. How to create and use such communicators is described at a later point in this chapter. For now, it suffices to use MPI_COMM_WORLD as the communicator argument to all the MPI functions that require a communicator.

6.3.3 Getting Information

The MPI_Comm_size and MPI_Comm_rank functions are used to determine the number of processes and the label of the calling process, respectively. The calling sequences of these routines are as follows: int MPI_Comm_size(MPI_Comm comm, int *size) int MPI_Comm_rank(MPI_Comm comm, int *rank)

The function MPI_Comm_size returns in the variable size the number of processes that belong to the communicator comm. So, when there is a single process per processor, the call MPI_Comm_size(MPI_COMM_WORLD, &size) will return in size the number of processors used by the program. Every process that belongs to a communicator is uniquely identified by its ***rank***. The rank of a process is an integer that ranges from zero up to the size of the communicator minus one. A process can determine its rank in a communicator by using the MPI_Comm_rank function that takes two arguments: the communicator and an integer variable rank. Up on return, the variable rank stores the rank of the process. Note that each process that calls either one of these functions must belong in the supplied communicator, otherwise an error will occur.

Example 6.2 Hello World

We can use the four MPI functions just described to write a program that prints out a "Hello World" message from each processor.

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
          myrank, npes);
  MPI_Finalize();
}
```

6.3.4 Sending and Receiving Messages

The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively. The calling sequences of these routines are as follows: int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

`MPI_Send` sends the data stored in the buffer pointed by `buf`. This buffer consists of consecutive entries of the type specified by the parameter `datatype`. The number of entries in the buffer is given by the parameter `count`. The correspondence between MPI datatypes and those provided by C is shown in Table 6.2. Note that for all C datatypes, an equivalent MPI datatype is provided. However, MPI allows two additional datatypes that are not part of the C language. These are `MPI_BYTE` and `MPI_PACKED`.

`MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data. Note that the length of the message in `MPI_Send`, as well as in other MPI routines, is specified in terms of the number of entries being sent and not in terms of the number of bytes. Specifying the length in terms of the number of entries has the advantage of making the MPI code portable, since the number of bytes used to store various datatypes can be different for different architectures.

The destination of the message sent by `MPI_Send` is uniquely specified by the `dest` and `comm` arguments. The `dest` argument is the rank of the destination process in the communication domain specified by the communicator `comm`. Each message has an integer-valued `tag` associated with it. This is used to distinguish different types of messages. The message-`tag` can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`. Even though the value of `MPI_TAG_UB` is implementation specific, it is at least 32,767.

`MPI_Recv` receives a message sent by a process whose rank is given by the `source` in the communication domain specified by the `comm` argument. The tag of the sent message must be that specified by the `tag` argument. If there are many messages with identical tag from the same process, then any one of these messages is received. MPI allows specification of wildcard arguments for both `source` and `tag`. If `source` is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message. Similarly, if `tag` is set to `MPI_ANY_TAG`, then messages with any tag are accepted. The received message is stored in continuous locations in the buffer pointed to by `buf`. The `count` and `datatype` arguments of `MPI_Recv` are used to specify the length of the supplied buffer. The received message should be of length equal to or less than this length. This allows the receiving process to not know the exact size of the message being sent. If the received message is larger than the supplied buffer, then an overflow error will occur, and the routine will return the error `MPI_ERR_TRUNCATE`.

Table 6.2. Correspondence between the datatypes supported by MPI and those supported by C.

MPI Datatype	C Datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`

MPI Datatype	C Datatype
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

After a message has been received, the `status` variable can be used to get information about the `MPI_Recv` operation. In C, `status` is stored using the `MPI_Status` data-structure. This is implemented as a structure with three fields, as follows: typedef struct MPI_Status { int MPI_SOURCE; int MPI_TAG; int MPI_ERROR; };

`MPI_SOURCE` and `MPI_TAG` store the source and the tag of the received message. They are particularly useful when `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are used for the `source` and `tag` arguments. `MPI_ERROR` stores the error-code of the received message.

The status argument also returns information about the length of the received message. This information is not directly accessible from the `status` variable, but it can be retrieved by calling the `MPI_Get_count` function. The calling sequence of this function is as follows: int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

`MPI_Get_count` takes as arguments the `status` returned by `MPI_Recv` and the type of the received data in `datatype`, and returns the number of entries that were actually received in the `count` variable.

The `MPI_Recv` returns only after the requested message has been received and copied into the buffer. That is, `MPI_Recv` is a blocking receive operation. However, MPI allows two different implementations for `MPI_Send`. In the first implementation, `MPI_Send` returns only after the corresponding `MPI_Recv` have been issued and the message has been sent to the receiver. In the second implementation, `MPI_Send` first copies the message into a buffer and then returns, without waiting for the corresponding `MPI_Recv` to be executed. In either implementation, the buffer that is pointed by the `buf` argument of `MPI_Send` can be safely reused and overwritten. MPI programs must be able to run correctly regardless of which of the two methods is used for implementing `MPI_Send`. Such programs are called ***safe***. In writing safe MPI programs, sometimes it is helpful to forget about the alternate implementation of `MPI_Send` and just think of it as being a blocking send operation.

Avoiding Deadlocks The semantics of `MPI_Send` and `MPI_Recv` place some restrictions on how we can mix and match send and receive operations. For example, consider the following piece of code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If `MPI_Send` is implemented using buffering, then this code will run correctly provided that sufficient buffer space is available. However, if `MPI_Send` is implemented by blocking until the matching receive has been issued, then neither of the two processes will be able to proceed. This is because process zero (i.e., `myrank == 0`) will wait until process one issues the matching `MPI_Recv` (i.e., the one with `tag` equal to 1), and at the same time process one will wait until process zero performs the matching `MPI_Send` (i.e., the one with `tag` equal to 2). This code fragment is not safe, as its behavior is implementation dependent. It is up to the programmer to ensure that his or her program will run correctly on any MPI implementation. The problem in this program can be corrected by *matching the order in which the send and receive operations are issued*. Similar deadlock situations can also occur when a process sends a message to itself. Even though this is legal, its behavior is implementation dependent and must be avoided.

Improper use of `MPI_Send` and `MPI_Recv` can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion. Consider the following piece of code, in which process *i* sends a message to process *i* + 1 (modulo the number of processes) and receives a message from process *i* - 1 (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

When `MPI_Send` is implemented using buffering, the program will work correctly, since every call to `MPI_Send` will get buffered, allowing the call of the `MPI_Recv` to be performed, which will transfer the required data. However, if `MPI_Send` blocks until the matching receive has been issued, all processes will enter an infinite wait state, waiting for the neighboring process to issue a `MPI_Recv` operation. Note that the deadlock still remains even when we have only two processes. Thus, when pairs of processes need to exchange data, the above method leads to an unsafe program. The above example can be made safe, by rewriting it as follows: 1 int a[10], b[10], npes, myrank; 2 MPI_Status status; 3 ... 4 MPI_Comm_size(MPI_COMM_WORLD, &npes); 5 MPI_Comm_rank(MPI_COMM_WORLD, &myrank); 6 if (myrank%2 == 1) { 7 MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD); 8 MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD); 9 } 10 else { 11 MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD); 12 MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD); 13 } 14 ...

This new implementation partitions the processes into two groups. One consists of the odd-numbered processes and the other of the even-numbered processes. The odd-numbered processes perform a send followed by a receive, and the even-numbered processes perform a receive followed by a send. Thus, when an odd-numbered process calls `MPI_Send`,the target process (which has an even number) will call `MPI_Recv` to receive that message, before attempting to send its own message.

Sending and Receiving Messages Simultaneously The above communication pattern appears frequently in many message-passing programs, and for this reason MPI provides the `MPI_Sendrecv` function that both sends and receives a message.

`MPI_Sendrecv` does not suffer from the circular deadlock problems of `MPI_Send` and `MPI_Recv`. You can think of `MPI_Sendrecv` as allowing data to travel for both send and receive simultaneously. The calling sequence of `MPI_Sendrecv` is the following: int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

The arguments of MPI_Sendrecv are essentially the combination of the arguments of MPI_Send and MPI_Recv. The send and receive buffers must be disjoint, and the source and destination of the messages can be the same or different. The safe version of our earlier example using MPI_Sendrecv is as follows.

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
             b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD, &status);
...
```

In many programs, the requirement for the send and receive buffers of MPI_Sendrecv be disjoint may force us to use a temporary buffer. This increases the amount of memory required by the program and also increases the overall run time due to the extra copy. This problem can be solved by using that MPI_Sendrecv_replace MPI function. This function performs a blocking send and receive, but it uses a single buffer for both the send and receive operation. That is, the received data replaces the data that was sent out of the buffer. The calling sequence of this function is the following: int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

Note that both the send and receive operations must transfer data of the same datatype.

6.3.5 Example: Odd-Even Sort

We will now use the MPI functions described in the previous sections to write a complete message-passing program that will sort a list of numbers using the odd-even sorting algorithm. Recall from Section 9.3.1 that the odd-even sorting algorithm sorts a sequence of *n* elements using *p* processes in a total of *p* phases. During each of these phases, the odd-or even-numbered processes perform a compare-split step with their right neighbors. The MPI program for performing the odd-even sort in parallel is shown in Program 6.1. To simplify the presentation, this program assumes that *n* is divisible by *p*.

Program 6.1 Odd-Even Sorting

[View full width]

```
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

main(int argc, char *argv[])
{
  int n;         /* The total number of elements to be sorted */
  int npes;      /* The total number of processes */
  int myrank;    /* The rank of the calling process */
  int nlocal;    /* The local number of elements, and the array that stores them */
  int *elmnts;   /* The array that stores the local elements */
  int *relmnts;  /* The array that stores the received elements */
  int oddrank;   /* The rank of the process during odd-phase communication */
  int evenrank;  /* The rank of the process during even-phase communication */
  int *wspace;   /* Working space during the compare-split operation */
  int i;
  MPI_Status status;

  /* Initialize MPI and get system information */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  n = atoi(argv[1]);
  nlocal = n/npes; /* Compute the number of elements to be stored locally. */

  /* Allocate memory for the various arrays */
```

```c
  elmnts  = (int *)malloc(nlocal*sizeof(int));
  relmnts = (int *)malloc(nlocal*sizeof(int));
  wspace  = (int *)malloc(nlocal*sizeof(int));

  /* Fill-in the elmnts array with random elements */
  srandom(myrank);
  for (i=0; i<nlocal; i++)
    elmnts[i] = random();

  /* Sort the local elements using the built-in quicksort routine */
  qsort(elmnts, nlocal, sizeof(int), IncOrder);

  /* Determine the rank of the processors that myrank needs to communicate during the */
  /* odd and even phases of the algorithm */
  if (myrank%2 == 0) {
    oddrank  = myrank-1;
    evenrank = myrank+1;
  }
  else {
    oddrank  = myrank+1;
    evenrank = myrank-1;
  }

  /* Set the ranks of the processors at the end of the linear */
  if (oddrank == -1 || oddrank == npes)
    oddrank = MPI_PROC_NULL;
  if (evenrank == -1 || evenrank == npes)
    evenrank = MPI_PROC_NULL;

  /* Get into the main loop of the odd-even sorting algorithm */
  for (i=0; i<npes-1; i++) {
    if (i%2 == 1) /* Odd phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
          nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
    else /* Even phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
          nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

    CompareSplit(nlocal, elmnts, relmnts, wspace,
                 myrank < status.MPI_SOURCE);
  }

  free(elmnts); free(relmnts); free(wspace);
  MPI_Finalize();
}

/* This is the CompareSplit function */
CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
             int keepsmall)
{
  int i, j, k;

  for (i=0; i<nlocal; i++)
    wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */

  if (keepsmall) { /* Keep the nlocal smaller elements */
    for (i=j=k=0; k<nlocal; k++) {
      if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
        elmnts[k] = wspace[i++];
      else
        elmnts[k] = relmnts[j++];
    }
  }
  else { /* Keep the nlocal larger elements */
    for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
      if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
```

```c
        elmnts[k] = wspace[i--];
      else
        elmnts[k] = relmnts[j--];
    }
  }
}

/* The IncOrder function that is called by qsort is defined as follows */
int IncOrder(const void *e1, const void *e2)
{
  return (*((int *)e1) - *((int *)e2));
}
```

6.5 Overlapping Communication with Computation

The MPI programs we developed so far used blocking send and receive operations whenever they needed to perform point-to-point communication. Recall that a blocking send operation remains blocked until the message has been copied out of the send buffer (either into a system buffer at the source process or sent to the destination process). Similarly, a blocking receive operation returns only after the message has been received and copied into the receive buffer. For example, consider Cannon's matrix-matrix multiplication program described in Program 6.2. During each iteration of its main computational loop (lines 47� 57), it first computes the matrix multiplication of the sub-matrices stored in `a` and `b`, and then shifts the blocks of `a` and `b`, using `MPI_Sendrecv_replace` which blocks until the specified matrix block has been sent and received by the corresponding processes. In each iteration, each process spends $O(n^3/p^{1.5})$ time for performing the matrix-matrix multiplication and $O(n^2/p)$ time for shifting the blocks of matrices *A* and *B*. Now, since the blocks of matrices *A* and *B* do not change as they are shifted among the processors, it will be preferable if we can overlap the transmission of these blocks with the computation for the matrix-matrix multiplication, as many recent distributed-memory parallel computers have dedicated communication controllers that can perform the transmission of messages without interrupting the CPUs.

6.5.1 Non-Blocking Communication Operations

In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations. These functions are `MPI_Isend` and `MPI_Irecv`. `MPI_Isend` starts a send operation but does not complete, that is, it returns before the data is copied out of the buffer. Similarly, `MPI_Irecv` starts a receive operation but returns before the data has been received and copied into the buffer. With the support of appropriate hardware, the transmission and reception of messages can proceed concurrently with the computations performed by the program upon the return of the above functions.

However, at a later point in the program, a process that has started a non-blocking send or receive operation must make sure that this operation has completed before it proceeds with its computations. This is because a process that has started a non-blocking send operation may want to overwrite the buffer that stores the data that are being sent, or a process that has started a non-blocking receive operation may want to use the data it requested. To check the completion of non-blocking send and receive operations, MPI provides a pair of functions `MPI_Test` and `MPI_Wait`. The first tests whether or not a non-blocking operation has finished and the second waits (i.e., gets blocked) until a non-blocking operation actually finishes.

The calling sequences of `MPI_Isend` and `MPI_Irecv` are the following: int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

Note that these functions have similar arguments as the corresponding blocking send and receive functions. The main difference is that they take an additional argument `request`. `MPI_Isend` and `MPI_Irecv` functions allocate a *request object* and return a pointer to it in the `request` variable. This request object is used as an argument in the `MPI_Test` and `MPI_Wait` functions to identify the operation whose status we want to query or to wait for its completion.

Note that the `MPI_Irecv` function does not take a `status` argument similar to the blocking receive function, but the status information associated with the receive operation is returned by the `MPI_Test` and `MPI_Wait` functions.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

`MPI_Test` tests whether or not the non-blocking send or receive operation identified by its `request` has finished. It returns `flag = {true}` (non-zero value in C) if it completed, otherwise it returns `{false}` (a zero value in C). In the case that the non-blocking operation has finished, the request object pointed to by `request` is deallocated and `request` is set to `MPI_REQUEST_NULL`. Also the `status` object is set to contain information about the operation. If the operation has not finished, `request` is not modified and the value of the `status` object is undefined. The `MPI_Wait` function blocks until the non-blocking operation identified by `request` completes. In that case it deal-locates the `request` object, sets it to `MPI_REQUEST_NULL`, and returns information about the completed operation in the `status` object.

For the cases that the programmer wants to explicitly deallocate a request object, MPI provides the following function.

```
int MPI_Request_free(MPI_Request *request)
```

Note that the deallocation of the request object does not have any effect on the associated non-blocking send or receive operation. That is, if it has not yet completed it will proceed until its completion. Hence, one must be careful before explicitly deallocating a request object, since without it, we cannot check whether or not the non-blocking operation has completed.

A non-blocking communication operation can be matched with a corresponding blocking operation. For example, a process can send a message using a non-blocking send operation and this message can be received by the other process using a blocking receive operation.

Avoiding Deadlocks By using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts. For example, as we discussed in Section 6.3 the following piece of code is not safe.

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

However, if we replace either the send or receive operations with their non-blocking counterparts, then the code will be safe, and will correctly run on any MPI implementation.

```
int a[10], b[10], myrank;
MPI_Status status;
MPI_Request requests[2];
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
```

```
   MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
 }
 ...
```

This example also illustrates that the non-blocking operations started by any process can finish in any order depending on the transmission or reception of the corresponding messages. For example, the second receive operation will finish before the first does.

Example: Cannon's Matrix-Matrix Multiplication (Using Non-Blocking Operations)

Program 6.3 shows the MPI program that implements Cannon's algorithm using non-blocking send and receive operations. The various parameters are identical to those of Program 6.2.

Program 6.3 Non-Blocking Cannon's Matrix-Matrix Multiplication

```
MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
                                 double *c, MPI_Comm comm)
 {
   int i, j, nlocal;
   double *a_buffers[2], *b_buffers[2];
   int npes, dims[2], periods[2];
   int myrank, my2drank, mycoords[2];
   int uprank, downrank, leftrank, rightrank, coords[2];
   int shiftsource, shiftdest;
   MPI_Status status;
   MPI_Comm comm_2d;
   MPI_Request reqs[4];

   /* Get the communicator related information */
   MPI_Comm_size(comm, &npes);
   MPI_Comm_rank(comm, &myrank);

   /* Set up the Cartesian topology */
   dims[0] = dims[1] = sqrt(npes);

   /* Set the periods for wraparound connections */
   periods[0] = periods[1] = 1;

   /* Create the Cartesian topology, with rank reordering */
   MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

   /* Get the rank and coordinates with respect to the new topology */
   MPI_Comm_rank(comm_2d, &my2drank);
   MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

   /* Compute ranks of the up and left shifts */
   MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
   MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

   /* Determine the dimension of the local matrix block */
   nlocal = n/dims[0];

   /* Setup the a_buffers and b_buffers arrays */
   a_buffers[0] = a;
   a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
   b_buffers[0] = b;
   b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));

   /* Perform the initial matrix alignment. First for A and then for B */
   MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
   MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
       shiftdest, 1, shiftsource, 1, comm_2d, &status);

   MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
   MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
       shiftdest, 1, shiftsource, 1, comm_2d, &status);

   /* Get into the main computation loop */
   for (i=0; i<dims[0]; i++) {
     MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
         leftrank, 1, comm_2d, &reqs[0]);
     MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
         uprank, 1, comm_2d, &reqs[1]);
     MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
         rightrank, 1, comm_2d, &reqs[2]);
     MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
         downrank, 1, comm_2d, &reqs[3]);

     /* c = c + a*b */
     MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

     for (j=0; j<4; j++)
       MPI_Wait(&reqs[j], &status);
   }

   /* Restore the original distribution of a and b */
   MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
   MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
       shiftdest, 1, shiftsource, 1, comm_2d, &status);

   MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
   MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
       shiftdest, 1, shiftsource, 1, comm_2d, &status);

   MPI_Comm_free(&comm_2d); /* Free up communicator */

   free(a_buffers[1]);
   free(b_buffers[1]);
}
```

There are two main differences between the blocking program (Program 6.2) and this non-blocking one. The first difference is that the non-blocking program requires the use of the additional arrays *a_buffers* and *b_buffers*, that are used as the buffer of the blocks of *A* and *B* that are being received while the computation involving the previous blocks is performed. The second difference is that in the main computational loop, it first starts the non-blocking send operations to send the locally stored blocks of *A* and *B* to the processes left and up the grid, and then starts the non-blocking receive operations to receive the blocks for the next iteration from the processes right and down the grid. Having initiated these four non-blocking operations, it proceeds to perform the matrix-matrix multiplication of the blocks it currently stores. Finally, before it proceeds to the next iteration, it uses `MPI_Wait` to wait for the send and receive operations to complete.

Note that in order to overlap communication with computation we have to use two auxiliary arrays � one for *A* and one for *B*. This is to ensure that incoming messages never overwrite the blocks of *A* and *B* that are used in the computation, which proceeds concurrently with the data transfer. Thus, increased performance (by overlapping communication with computation) comes at the expense of increased memory requirements. This is a trade-off that is often made in message-passing programs, since communication overheads can be quite high for loosely coupled distributed memory parallel computers.