**3.1- Wormhole Routing in Mesh Networks**

An n-dimensional mesh is defined as the interconnection structure that has K0 x K1 x…….xKn-1 nodes, where n is the number of dimensions of the network and Ki is the radix of dimension i. Each node is identified by an n-coordinate vector (x0, x1, . . . , xn-1), where 0 ≤ xi ≤ Ki - 1. A number of routing techniques have been used for mesh networks. These include *dimension-ordered, dimension reversal, turn model,* and *message flow model*. In the following, we introduce the dimension-ordered of X-Y routing.

**Dimension-Ordered (X-Y) Routing** A channel numbering scheme often used in n-dimensional meshes is based on the dimension of channels. In dimension-ordered routing, each packet is routed in one dimension at a time, arriving at the proper coordinate in each dimension before proceeding to the next dimension. By enforcing a strict monotonic order on the dimensions traversed, deadlock-free routing is guaranteed. In a two-dimensional mesh, each node is represented by its position (x, y); the packets are first sent along the x-dimension and then along the y-dimension, hence the name X-Y routing.

In X-Y routing, messages are first sent along the X-dimension and then along the Y-dimension. In other words, at most one turn is allowed and that turn must be from the X-dimension to the Y-dimension. Let (sx, sy) and (dx, dy) denote the addresses of a source and destination node, respectively. Assume also that (gx, gy) = (dx - sx, dy - sy). The X-Y routing can be implemented by placing gx and gy in the first two flits, respectively, of the message. When the first flit arrives at a node, it is decremented or incremented, depending on whether it is greater than 0 or less than 0. If the result is not equal to 0, the message is forwarded in the same direction in which it arrived. If the result equals 0 and the message arrived on the Y-dimension, the message is delivered to the local node. If the result equals 0 and the message arrived on the X-dimension, the flit is discarded and the next flit is examined on arrival. If that flit is 0, the packet is delivered to the local node; otherwise, the packet is forwarded in the Y-dimension. Figure 5.7 shows an example of the X-Y routing between a source node and a destination node in an 8 x 8 mesh network.



Figure 5.7 Dimension-ordered (X-Y) routing in an 8 x 8 mesh network.

**3.2- Virtual Channels**

The principle of virtual channel was introduced in order to allow the design of deadlock-free routing algorithms. Virtual channels provide an inexpensive method to increase the number of logical channels without adding more wires. A number of adaptive routing algorithms are based on the use of virtual channels.

A network without virtual channels is composed of single lane streets. Adding virtual channels to an interconnection network is analogous to adding lanes to a street network, thus allowing blocked messages to be passed. In addition to increasing throughput, virtual channels provide an additional degree of freedom in allocating resources to messages in a network. Consider the simple network shown in Figure 5.8.



Figure 5.8 Path multiplexing through the same link.

In this case, two paths X-A-B-Z and Y-A-B-W share the common link AB. It is, therefore, required to multiplex link AB between the two paths (two lanes). A provision is also needed such that data sent over the first path (lane) is sent from X to Z and not to W and similarly data sent over the second path (lane) is sent from Y to W and not to Z. This can be achieved if we assume that each physical link is actually divided into a number of unidirectional virtual channels. Each channel can carry data for one virtual circuit (one path). A circuit (path) from one node to another consists of a sequence of channels on the links along the path between the two nodes.

When data is sent from node A to node B, then node B will have to determine the circuit associated with the data such that it can decide whether is should route the data to node Z or to node W. One way that can be used to provide such information is to divide the AB link into a fixed number of time slots and statically assign each time slot to a channel. This way, the time slot on which the data arrives identifies the sending channel and therefore can be used to direct the data to the appropriate destination.

One of the advantages of the virtual channel concept is deadlock avoidance. This can be done by assigning a few flits per node of buffering. When a packet arrives at a virtual channel, it is put in the buffer and sent along the appropriate time slot.

**4- Message Passing Programming Models**

A message passing architecture uses a set of primitives that allows processes to communicate with each other. These include the *send, receive,* *broadcast*, and *barrier* primitives. The **send** primitive takes a memory buffer and sends it to a destination node. The **receive** primitive accepts a message from a source node and stores it in a specified memory buffer. The basic programming model used in message passing architectures is based on the idea of matching a send request on one processor with a receive request on another. In such scheme, send and receive are blocking; that is, send blocks until the corresponding receive is executed before data can be transferred.

Implementation of the send/receive among processes requires a three-way protocol as shown in Figure 5.9. In this case, the sending process issues a request-to-send message to the receiver process. The latter stores the request and sends a reply message back. When the corresponding receive is executed, the sender process receives the reply and finally transfers the data. The blocking send/receive is simple; it requires no buffering at the source or the destination. However, the three-way handshaking used in blocking send/receive requires that both the sender and the receiver be blocked for at least a full round-trip time. During this time the processors are idle, thus leading to an increase in the network communication latency. In addition, with blocking send/receive, it is impossible to overlap communication with computation and thus the network bandwidth cannot be fully utilized.



Figure 5.9 Blocking send/receive handshaking protocol.

The use of nonblocking operation is utilized by most message passing implementations in order to avoid the drawbacks of the three-phase protocol. In this case, send appears immediately to the user program. However, the message is buffered by the message layer until the network port is available. Only then, would the message be transmitted to the recipient. In there, the message is again buffered until a matching *receive* is executed.

Table 5.2 shows the performance of the send/receive on a number of message passing machines. In this table, Ts represents the message start-up cost, Tb represents the per-byte cost, and Tfp is the average cost of a floating-point operation. It should be noted that the CM-5 is blocking and uses a three-phase protocol. The *iPSC* long messages also use a three-phase protocol in order to guarantee that enough buffer space is available at the receiving node.

TABLE 5.2 Performance of Send/Receive on a Number of Message Passing Machines



**Example 2** Let us try to compute y = (a ‏+ b) \* (c ‏+ d) on a single processor and on a message architecture consisting of two processors. In presenting illustrative solutions to this problem, we will use simple and self-explanatory notations.

**(a) Using Single Processor**



**(b) Message Passing With Two Processors P1 and P2**

Assume that the operands are distributed between the two processors as shown.





**Example 3** It is required to sum all components of a vector A, having n components (for simplicity assume that n is a power of 2) using p processors, assuming that the vector components are distributed among the p processors.

. Initialization step: Each processor performs the partial sum of the vector components it has.

. Repeat using index k = 1 to n/2 in powers of 2.

. Processor j and processor j ‏+ k send and receive data in pairwise fashion and perform the summation.

Figure 5.10 shows how the process can be performed in log2 n steps.



Figure 5.10 Summation in log2 n steps using a message passing system.

**5- Processor Support for Message Passing**

Processors that support message passing are those processors that contain the special instructions needed to support interprocess message communications. In order to support interprocess communications, a number of features are required. Among these, the following features are needed:

1. A port is a communication channel. It is a reference object for tasks and threads. Two main operations can be performed on ports: send and receive.

2. Messages are used as communication among objects. A message is divided into a header and a body. The size of the body is variable while that of the header is fixed. A message holds information exchanged between processes.

3. Port sets: A task can hold multiple access rights (send and receive) on ports. Multiple tasks can hold send access to a single port. On the other hand, one task can hold receive access at a given time. In port set, a task can have either all or none of the access rights to a group of ports. Ports must be mutually exclusive in the sense that a port cannot be in two different sets at a given time.

The Intel iPAX 432 uses message passing communications and supports them directly. It also uses port objects that work as a competitor to the path of the message. The processor contains a message queue. A message communication can be arranged depending on the following:

1. Time of arrival (such as the “first-in-first-out”, FIFO);

2. Priority;

3. Deadline within priority.

The iPAX 432 produces a nonblocking message passing by using specific process that has conditional SEND and RECEIVE operations. The operand of these conditional operations is a specific Boolean flag. Thus, if the unconditional operation corresponding to the conditional one was blocking, the conditional operation result will be false and if it is nonblocking then the conditional operation will be true. In this case of the conditional operations, to support message passing, a correct communication and interaction between and within each process must be satisfied. This must be because the processor will continue executing a specific operation in case it cannot complete the communication operation. Therefore, a good program must be able to decide whether to retry the operation by testing the returned flag.

There are also other kinds of message passing operations that are not blocking, such as the SURROGATE-SEND and SURROGATE-RECEIVE. These operations hold the operation in a waiting queue and are sufficient for use with high-level language interprocess communication. The operand of these operations is an event called DONE. These operations do the send and receive operations and put the message in the port’s queue. The end or the completion of these operations has DONE (event) when the SURROGATE received the desired service. The original process is responsible for checking the completion of the operation by searching for the DONE event.

The IBM AS/400 supports message passing by having an event object type that contains a field supporting the contents of the message. This field is called the event data field. AS/400 processor operations are *send* and *receive*. The send operation is used to send the interprocess message by a processor operation called SIGNALEVENT (PROC, EV, DATA). This processor operation has three parameters. The first two are essential to exist such that the event EV will be signaled within the processor PROC. The third parameter is unessential for that event. To receive the interprocess message, the WAIT-ON-EVENT, TEST-EVENT, MONITOREVENT, and RETRIEVE-EVENT-DATA are used. Note that an exactly blocking receive operation does not exist because the value of the timeout should be determined with every operation that might block the execution of a process.

**6- Message Passing Versus Shared Memory Architectures**

Shared memory enjoys the desirable feature that all communications are done using implicit loads and stores to a global address space. Another fundamental feature of shared memory is that synchronization and communication are distinct. Special synchronization operations (mechanisms), in addition to the loads and stores operations, need to be employed in order to detect when data have been produced and/or consumed. On the other hand, message passing employs an explicit communication model. Explicit messages are exchanged among processors. Synchronization and communication are unified in message passing.

The generation of remote, asynchronous events is an integral part of the message passing communication model. It is important, however, to indicate that shared memory and message passing communication models are universal; that is, it is possible to employ one to simulate the other. However, it is observed that it is easier to simulate shared memory using message passing than the converse. This is basically because of the asynchronous event semantics of message passing as compared to the polling semantics of the shared memory.

A number of desirable features characterize shared memory architectures. The shared memory communication model allows the programmer to concentrate on the issues related to parallelism by relieving him/her of the details of the interprocessor communication. In that sense, the shared memory communication model represents a straightforward extension of the uniprocessor programming paradigm. In addition, shared memory semantics are independent of the physical location and therefore they are open to the dynamic optimization offered by the underlying operating system. On the other hand, the shared memory communication model is in essence a polling interface. This is a drawback as far as synchronization is concerned. This fact has been recognized by a number of multiprocessor architects and their response has always been to augment the basic shared memory communication model with additional synchronization mechanisms. An additional drawback of shared memory is that in order for data to cross the network, a complete round trip has to be made. One-way communication of data is not possible.

Message passing can be characterized as employing an interrupt-driven communication model. In message passing, messages include both data and synchronization in a single unit. As such, the message passing communication model lends itself to those operating system activities in which communication patterns are explicitly known in advance, for example, I/O, interprocessor interrupts, and task and data migration. The message passing communication model lends itself also to applications that have large synchronization components, for example, solution of systems of sparse matrices and event-driven simulation. In addition, message passing communication models are natural client–server style decomposition. On the other hand, message passing suffers from the need for *marshaling cost*, that is, the cost of assembling and disassembling of the message.

One natural conclusion arising from the above discussion is that shared memory and message passing communication models each lend themselves naturally to certain application domains. Shared memory manifests itself to application writers while message passing manifests itself to operating systems designers. It is therefore natural to consider combining both shared memory and message passing in general purpose multiprocessor systems. This has been the main driving force behind systems such as the Stanford FLexible Architecture for SHared memory (FLASH) system. It is a multiprocessor system that efficiently integrates support for shared memory and message passing while minimizing both hardware and software overhead.