INTERCONNECTION NETWORKS

A parallel computer architecture can be viewed as a collection of processors that cooperate and communicate to solve a problem. This abstract view of parallel architectures is true for both SIMD and MIMD parallel architectures. Processors can communicate to share data or to synchronize program execution. There are two mechanisms to support inter-processor communication: (1) a shared memory or (2) an interconnection network. Parallel architectures that use the first scheme are called *shared memory* parallel architecures and those using the second scheme are called *distributed memory* parallel architectures. In a distributed memory architecture each processor has its own local memory and processors communicate explicity through the interconnection network to access the data stored in each other's local memory. We shall focus on distributed memory architectures due to their popularity and commercial abundance. This section discusses the salient aspects of interconnection networks that apply to both SIMD and MIMD architectures.

The time required for communication, called the communication time, increases the overall execution time of the algorithm. The amount, and pattern, of communication depends on the application and the algorithm used to solve the problem. Since all communication is done through the interconnection network, it is one of the most critical components of a parallel processor. Some important performance criteria for interconnection networks are:

1 Latency: the transit time for a single message.

2 Bandwidth: the amount of message traffic the network can support.

3 Cost of Hardware.

4 Diameter: the maximum distance between processors.

Interconnection networks can be broadly categorized based on control policy (distributed vs central), switching policy (packet switching vs circuit switching), synchronous versus asynchronous operation, and network topology (static topology vs dynamic topology). We first define the control and switching policies and then study some interconnection networks in the two classes of topologies.

Switching Policy: There are two methods for routing messages between processors through the interconnection network - packet switching and circuit switching. In packet switching a message is sent to its destination node through a series of store-and-forward steps without establishing a physical path between source and destination processors. In circuit switching, a circuit (path) is established on the network between the source and destination processors following which the entire message is placed on the path. While circuit switching reduces the time to transfer data between two non-neighboring processors it also includes a substantial overhead in terms of reduced bandwidth and control algorithms.

In packet switching a processor can send a message to one its neighboring processors; i.e., a SEND instruction must specify which of its "output ports" (i.e., network connections) must be used. In general, a message consists of a number of fields: the destination processor address, source

address, length of message (no. of bytes) and the message itself. In packet switching systems each message is broken up into a number of packets each of size "packet size"; for example, if the packet size is four bytes then a message of 16 bytes is broken into four packets. Each packet is sent to a processor which then proceeds to forward the packet to the next processor in the path. A routing startegy in packet switching need only specify the next processor in the path; in circuit switching it determines a path and then sets up a physical circuit (and the entire message is sent along this physical path).

Control Policy: The interconnection functions that have to be realized by setting the switches can be set by central control or control that is distributed among the processors. For e.g., SIMD architectures use central control with the CU setting the desired switches.

Synchronous vs Asynchronous Operation: In synchronous mode the network operates in a synchronous mode so that all processors run on the same clock and perform a data transfer step (communication step) in tandem. In asynchronous mode, each processor operates on its own clock and processors do not synchornize their communication steps.

NOTE: In the immediate future we shall assume synchronous systems (with centralized control) since our focus is on SIMD type architectures. Therefore, in a SEND cycle all (or some) processors do a Send and in the next cycle when a Receive instruction is executed it is assumed that the message will have been received by the neighbor processor.

Static vs Dynamic Topology: A static topology interconnection network is one that does not change its connections once built. In a dynamic topology network the connections can be changed by setting the appropriate switches. Two processors can communicate directly if there is a communication link between them. The set of processors that can communicate directly depends on the specific topology of the interconnection network. If there is no direct link between two processors then they communicate by sending messages along links in a path between them. The length of the path determines the amount of time it takes for the processors to communicate, and this length in turn is governed by the topology of the interconnection network. Therefore, the topology of the interconnection network is an important aspect of parallel architectures. Many interconnection networks with static and dynamic topologies have been designed and built. In what follows we shall discuss some common topologies and the routing strategies, *i.e.,* algorithms for sending messages between processors, used in these networks. We conclude this section with a simple example illustrating the need for communication (*i.e.,* data transfers) and the effect of the topology on the communication time.

Static Topology Interconnection Networks

A static topology interconnection network is appropriate for problems whose communication patterns are regular and can be predicted reasonably well, and that can be divided into parts that have highly 'local' exchanges (i.e. data exchanges occur mostly between neighbours). Applications that are well suited for this topology are ones that involve analysis of events in space, such as

weather modeling, where the data is represented as a two dimensional array of "grid points" and interaction is among neighboring grid points. Some examples of static topology are the mesh (the mesh includes the linear array, which is a one dimensional mesh), the ring, the star, the tree, the hypercube (also known as the cube), and the complete interconnection network. Considerations for selecting a network among these include cost, performance, and ease of routing messages. For example, the complete interconnection network has high performance, but also has the most hardware cost. Quite often, a tradeoff exists between cost and performance. As examples of static topology we discuss the mesh and hypercube topologies.

Routing on Interconnection Networks: A routing algorithm, for a given topology, determines the set of links to be used to route data between a pair of given processors. We shall discuss point-to-point routing strategies, *i.e.,* routing from a processor with address A to a processor with address B, which determine the shortest path between the pair of processors. It must be noted that conflicts for links (/edges) must be avoided when we design the routing algorithm. Also note that in graph theoretic terms, a route is essentially the shortest path between two nodes. While a number of different routing requirements arise, we shall focus on three types of routing: One-to-One (when a single processor has to send a message to one other processor), One-to-All/Broadcasting (when a single processor must send the same message to all processors) and All-to-One (when a processor must accumulate distinct messages from all other processors).

Model and Notations: We first define the model, and the notations, under which the routing algorithms (described in what follows) operate.

Each processor has a switch (i.e., I/O port) that connects to the Interconnection Network (ICN). The figure below illustrates this point; at each switch we have two "registers" R_{in} and R_{out} (or also called DTR_{in} and DTR_{out}): R_{in} stores the data *received* from the network and when a processor sends a message the contents of R_{out} are sent into the network.

We have two instructions $SEND$ and REC which send and receive data respectively. A SEND (or REC) instruction must specify the neighbor (i.e., I/O port) to which the message must be sent (received). Furthermore, a SEND always sends the data in the R_{out} register. Therefore, to send the contents of register R_0 a processor must first do "`LOAD` $R_0 \rightarrow R_{out}$" and then do a `SEND` neighbor. Similarly, if the incoming message has to be received and stored in register R_0 then proc must do "`REC` neighbor" and then "`LOAD` $R_{in} \rightarrow R_0$". (We shall henceforth use $A := B$ to mean `LOAD` contents of A into B.)

Furthermore, since we are assuming SIMD systems we must write routing algorithms in which *all* processors execute the same instruction. To this end, we assume that the statement "ENABLE processors with address X" **only** enables processors with address field X; this process is called *processor masking* and we shall discuss its implementation when we discuss SIMD architectures. At the current time we simply assume that masking schemes are provided to us and are specified by the instruction "ENABLE ..." (i.e., all processors that do not satisfy the condition are disabled and therefore do not execute the following instruction). For example, the instructions "ENABLE even-numbered processors" "SEND to-right-neighbor" specify that only processors with even address are enabled and they send a message to their right-neighbors.

Linear Array and Mesh Interconnection Networks

The simplest interconnection function is a one-dimensional mesh, or linear array, where the processors are arranged as a linear array and adjacent processors are connected. By connecting the two end processors we have a Ring network. In a linear array/ring each of the N processors is assigned an address i, for $0 \leq i \leq N-1$.

To route a message from processor i to processor j we simply forward the message Right (if $j > i$) or Left (if $i > j$). In SIMD systems this is equivalent to the following routing algorithm:

Routing Algo (point to point) From P_i to P_j (Central control)

If $j > i$ then

begin /* send message right $(j-i)$ steps */

For $k := i$ to $(j-1)$ do/* initially message is in R_{in} of P_i */

ENABLE P_k /* enable only P_k */

Send-Right (i.e., Send $P_k \rightarrow P_{k+1}$)

ENABLE P_{k+1}

Rec-Left (i.e., Rec $P_k \leftarrow P_{k-1}$)

$R_{out} := R_{in}$

endfor

Else (i.e., if $i > j$) then

Send-Left /* similar to above, but send left */

.....

endif

One-to-All or Broadcasting: In this routing scheme a message residing at a processor i must be sent to all processors in the system. For this routing, the algorithm simply forwards the message $(N-1)-i$ times to the right and i times to the Left. This algorithm is outlined below:

Routing Algo One-to-All (Linear Array): Broadcast from P_i

begin /* message is initially in R_{out} of P_i */

For $k := i$ to $(N-2)$ do

begin /* forward message to the right */

ENABLE P_k
Send-Right (i.e., Send $P_k \rightarrow P_{k+1}$)
ENABLE P_{k+1}
Rec-Left (i.e., Rec $P_k \leftarrow P_{k-1}$)
$R_{out} := R_{in}$
endfor
For $k := i$ downto 1 do /* send leftwards */
begin
ENABLE P_k
Send-Left (i.e., $P_k \rightarrow P_{k-1}$)
ENABLE P_{k-1}
Rec-Right (i.e., $P_k \leftarrow P_{k+1}$)
$R_{out} := R_{in}$
endfor
At end of algo, message is in R_{in} of every processor
end

All-to-One: In this problem a single processor must accumulate $N-1$ messages - one from each of the other processors. Firstly note that regardless of the type of network the lower bound for this is $N-1$ steps since a processor can send or receive only one message in one step.

Assume that processor 0 has to accumulate all the messages. This process can be accomplished in $N-1$ steps by having each processor send its message left. All processors (except processor 0) forward the message left one step. The following algorithm outlines the above process:

All-to-One Routing (Linear Array): All proc to P_i
Results are stored in array $A[j], 0 \leq j \leq N-1$ in memory of P_i
Initially data is in R_{out} of each processor
For $k := N-1$ downto $(i+1)$ do
begin /* fetch from processors to right */
ENABLE **all** processors P_x with $x \leq k$
Send-Left ($P_x \rightarrow P_{x-1}$)
ENABLE **all** proc. P_x where $i \leq x \leq k-1$
Rec-Right ($P_x \leftarrow P_{x+1}$)
$R_{out} := R_{in}$
ENABLE P_i /* in proc i store into array $A[\ell]$ */
$\ell := (N-1) - k + i + 1$
$A[\ell] := R_{in}$
endfor

For $k := 0$ to $i-1$ do /* rec. from left */

for all P_x where $k \leq x \leq i-1$

Send-Right

Rec-Left

in Proc P_i: $A[i-k-1] := R_{in}$

/* similar to previous loop */

end

Mesh Topology:

A k-dimensional mesh topology can be constructed analogously, where the processors are arranged as a k-dimensional array and each processor is connected to its neighboring array points in each of the k dimensions. Each processor is assigned a k-tuple address $(a_1, a_2, \ldots, a_n)$ which corresponds to the location of the processor in the k-dimensional space. Each interior processor in a k-dimensional mesh is connected to $2k$ other processors. There are many options for connecting processors at the boundaries and one scheme is to connect the processors at opposite boundaries, thus resulting in a *Torus* topology. In a mesh topology the routing algorithm simply routes along each dimension.

Point-to-Point Routing: Consider a two-dimensional $n \times n$ Mesh, with processor addresses $(i,j), 0 \leq i,j \leq n-1$ and $N = n^2$. Each processor is given a pair of addresses – the row number and the column number. To route from processor (i,j) to (k,l) the message is sent along row i from (i,j) to (i,l) and then along the column from (i,l) to (k,l); note that these steps are similar to the one-dimensional array routing. For example, to send data from processor $A = (0,2)$ to processor $B = (2,3)$ we first send to processor $(2,2)$ by sending data up through the column and then send from $(2,2)$ to $(2,3)$ along the row. This simple routing strategy is one of the advantages of the mesh. The routing algo is described below:

Routing 2-D Mesh (without wraparound) from $P_{i,j}$ to $P_{k,l}$

Message is in R_{out} of $P_{i,j}$

If $l > j$ then /* send right/east */

For $x := j$ to $l-1$ do

ENABLE $P_{i,x}$

Send-East ($P_{i,x} \rightarrow P_{i,x+1}$)

ENABLE $P_{i,x+1}$

Rec-West ($P_{i,x} \leftarrow P_{i,x-1}$)

$R_{out} := R_{in}$

endfor

else

For $x := j$ downto $l+1$ do

ENABLE $P_{i,x}$

Send-West ($P_{i,x} \rightarrow P_{i,x-1}$)

Rec-East

endfor

If $k > i$ then

Send southwards $(k - i)$ steps

else

Send northwards $i - k$ steps

end

All-to-One: The routing algorithm is relatively simple for Meshes: broadcast along each dimension. For 2-D Mesh to broadcast from (i, j) to all processors we first broadcast within the row (using the 1-D Mesh algorithm) so that the message is sent to all processors (x, j) for $0 \leq x \leq n-1$; then *in parallel* in each column we do a broadcast. For example, to broadcast from $(1, 2)$ to all: first step we broadcast from $(1, 2)$ to processors $(1, 0), (1, 1), (1, 3)$. In the second step, in columns $0, 1, 2, 3$ simultaneously we broadcast in the columns; i.e., $(1, 0)$ sends to $(2, 0)$, $(1, 1)$ sends to $(2, 1)$, $(1, 2)$ sends to $(2, 2)$ and $(1, 3)$ sends to $(2, 3)$ (and all this is done in one cycle). The complexity for step 1 is $n - 1$ steps and step 2 also takes $n - 1$ (for Mesh without wraparound but this is also the same for mesh with wraparound if we consider MIMD systems).

Broadcast 2-D Mesh: from $P_{i,j}$ to all

/* first broadcast in row i using 1-D algo */

/* then in parallel broadcast in all columns */

For $k := j$ to $n - 2$

ENABLE $P_{i,k}$

Send-East

ENABLE $P_{i,k+1}$

Rec-West

$R_{out} := R_{in}$

endfor

For $k := j$ downto 1 do

ENABLE $P_{i,k}$

Send-West

ENABLE $P_{i,k-1}$

Rec-East

$R_{out} := R_{in}$

For $k := i$ to $n - 2$ do

ENABLE bf all proc. $P_{k,x}$ for $0 \leq x \leq n - 1$

Send-South

ENABLE **all** proc $P_{k+1,x}$ for $0 \leq x \leq n-1$

Rec-north

$R_{out} := R_{in}$

endfor

For $k := i$ downto 1 do

send-north

rec-south

end.

All-to-One: To solve this problem we can take two approaches – (1) simulate the linear array on the mesh or (2) accumulate in one dimension at a time. In a 2-D Mesh where all messages must be accumulated in proc $(0,0)$; in the first step messages in all processors in the same column are accumulated into the processor in row 0 in that column. For example, all messages in column 0 (in processors $(1,0),(2,0),(3,0)$ are accumulated into proc $(0,0)$). This process is done in all columns concurrently and takes $n-1$ steps. In the second step, each processor in row 0 sends its n messages to processor $(0,0)$. This takes $n(n-1)$ steps.

Hypercube Interconnection Network

The hypercube is one of the most popular static topologies. A hypercube of dimension n, called an n-cube, consists of 2^n nodes, where each node represents a processor (with its local memory). The processors are numbered from 0 to $2^n - 1$ and are given an n-bit binary address corresponding to their number. Thus processor 3 in a 3-cube is given the address 011. Two processors are connected by a link if their binary addresses differ in exactly one bit position. The link connecting processors with addresses differing in the i-th bit position is called the i-th dimension link. For example, when $n = 3$, the processor with address 010 is connected to the processors 011, 000 and 110. The Figure illustrates hypercubes of dimensions 2 and dimensions 3. Hypercubes can be defined recursively; an $(n+1)$-cube is constructed from two n-cubes by (1) prefixing the addresses of processors in one hypercube with a 0 (to get the zero cube) and the addresses in the other with a 1 (to get the one cube) and (2) each of the processors in the zero cube is connected to its counterpart in the one cube. The distance between any two processors is defined by the hamming distance between their addresses, and thus we see that the maximum distance between any two processors (i.e., the diameter) is n for an n-cube which has 2^n processors.

Point-to-point routing: on the hypercube can be performed by using a simple algorithm. The shortest path between any two processors can be determined by performing an exclusive OR of their addresses. This leads to a simple routing policy; perform an exclusive OR of the source address A and destination address B and route along dimension i if the i-th bit of the exclusive OR operation is a 1. For example, consider the transfer of data from processor 010 to processor 111. The exclusive OR gives 101 and thus we need to route along dimension 0 link and dimension 2 link. This results in a data transfer from 010 to 011 in the first step and from 011 to 111 in

the second step. In general, the routing algorithm can be designed to work in a specific dimension sequence; for example it could send along dimension 0 first and then dimension 1 etc. etc.

One-to-All: Broadcasting in a hypercube is accomplished by exploiting the recursive structure of the hypercube. The processor with the message first sends a message to one of its neighbors. At the second step both processors send to neighbors in one dimension and this process is repeated. Essentially, a broadcast is performed in cubes of dimension 0, $1, ..., i, i+1, ...$

Broadcast-Hypercube: From $P_{a_{n-1}a_{n-2}..a_1a_0}$ to all
note: $\times$ (don't care) means bit is 0 or 1
For $j := 0$ to $n-1$
    ENABLE $P_{a_{n-1}a_{n-2}...a_j \times \times ... \times}$
    Send-Dimension j: $P_{a_{n-1}...a_j \times ... \times} \rightarrow P_{a_{n-1}...\bar{a}_j \times ... \times}$
    ENABLE **all** $P_{a_{n-1}...\bar{a}_j \times ... \times}$
    Rec-dimension j
    $R_{out} := R_{in}$
endfor

For example, consider broadcasting from processor with address 0000. First we send from 0000 to 0001 (in dimension 0). Second step, 0000 sends to 0010 and 0001 sends to 0011. Third step: 0000 to 0100, 0010 to 0110, 0001 to 0101 and 0011 to 0111. Fourth step: 0000 to 1000, 0100 to 1100, 0010 to 1010, 0110 to 1110, 0001 to 1001, 0101 to 1101, 0011 to 1011 and 0111 to 1111. Thus, the time needed is n for a n-dimensional hypercube.

For All-to-One routing, the hypercube can be arranged as a linear array; to do this simply provide a gray code sequence of the processor addresses. Thus, the linear array algorithm can be adapted with time $2^n - 1$ for a n-dimensional hypercube.