2.6.2 Sending m_{NS} Messages on Wormhole Routing Interconnection Networks

Above we state that a processor p sends a m_{NS} marker along a link when it determines that no more cross-cut messages will be sent along that link. However, we do not state specifically how p makes this determination. The example in Figure 2.7 was simple enough to provide an *ad hoc* solution, but here we consider the general case.

We are aided by the fact that while the interconnects on wormhole-routing multicomputers provide the illusion of full connectivity, in reality each of the $O(n^2)$ paths between processors corresponds to a route through some subset of the physical links of the system. Furthermore, with the exception of the original store-and-forward hypercube multicomputers, all of the multicomputer interconnection networks of which we are aware satisfy the following three conditions:

1. Routing in the system is deterministic, using a deadlock-free routing algorithm as defined by Dally and Seitz [DS87].

2. Allocation of links along a route is non-preemptive; that is, once a message is allocated a link along its route, no other message travels along the same link until the message is on the next link in the route (or until the message is at the destination processor).

3. Queueing for incoming messages in the system is strictly FIFO.

We call an interconnection network that satisfies these conditions a *well-behaved* interconnection network. Many existing and proposed multicomputers are well-behaved. Among these are Intel's iPSC/2, iPSC/860, Delta, Sigma and Paragon machines, Stanford's DASH [LLG+90], and MIT's J-Machine [Dal91b]. The first two of these use hypercube interconnections, while the last four are two and three-dimensional meshes.

For well-behaved interconnection networks, we present a method for sending the m_{NS} messages. We prove that the method is guaranteed to start and

terminate, and that it does what it promises: When m_{NS} is received on a link, no more cross-cut messages will be received on that link. Afterwards, we display how the method works for hypercubes and two-dimensional, toroidal meshes. Since a constant number of m_{NS} messages are sent per physical link of the interconnection network, the total number of m_{NS} messages is $O(l(n))$.

Method for General Well-behaved Interconnection Networks

Dally and Seitz [DS87] define an interconnection network I to be a strongly connected, directed graph $I = G(N, C)$, where N is the set of nodes, and C is the set of channels, or physical links that connect the nodes. A routing function $f : C \times N \rightarrow C$ maps the current channel c_i upon which a message resides, and its destination node n_d, to the next channel c_j that it will take on its way to n_d. Thus, f is a non-adaptive, memoryless routing function. It is non-adaptive because it always yields the same path from one node to another, and it is memoryless because the function works from channel to channel, retaining no knowledge of the source node once it is underway.

To prove a routing function deadlock-free, they define a *channel dependency graph* $D = G(C, E)$, where C is the same set of channels described above, and E is a set of edges determined by f as follows:

$$E = \{(c_i, c_j) \mid f(c_i, n_d) = c_j \text{ for some } n_d \in N\}$$

They show that f is deadlock-free if and only if D has no cycles.

We use the results of Dally and Seitz to define how to send the m_{NS} marker messages. It is assumed that each processor p_i resides on node n_i. Moreover, if the interconnection network contains the channel $c = (n_i, n_j)$, then the path of a message from p_i to p_j is routed through c. Finally, when we say "p_i sends m_{NS} on channel c," we mean that if $c = (n_i, n_j)$, then p_i sends m_{NS} to p_j, thereby using channel c. The method is quite simple:

- Assume that p_i has checkpointed itself.
- For all outgoing channels c_o from p_i, if there is no c_j such that $(c_j, c_o) \in E$, then p_i sends the marker m_{NS} on channel c_o.

- For all other outgoing channels c_o from p_i, when m_{NS} markers have been received from all channels c_j such that $(c_j, c_o) \in E$, p_i sends m_{NS} on channel c_o.

The following theorems prove the method's correctness:

Theorem 2.6.5 *Until m_{NS} messages have been sent on all channels, there is always some processor that may send a m_{NS} message.*

Proof: Since D is a graph with no cycles, we can number the channels from 1 to $|C|$ in such a way that if $(c_i, c_j) \in E$, then $i < j$. Now, since there is no c_i such that $(c_i, c_1) \in E$, a m_{NS} marker can instantly be sent on c_1. The proof proceeds by induction. Suppose that m_{NS} has been sent on channels c_1 through c_k, but not on channel c_{k+1}. Consider all c_i such that $(c_i, c_{k+1}) \in E$. From our numbering, $i < k + 1$, so m_{NS} has been sent out all channels c_i. As f is deadlock-free, the markers are eventually received, and m_{NS} can be sent out c_{k+1}. Therefore, there is always a channel that is ready to receive a m_{NS} marker. □

Theorem 2.6.6 *When m_{NS} is received on a channel c, it is guaranteed that no more cross-cut messages will be sent or received along c.*

Proof: Suppose p sends a cross-cut message m to p' (as p_2 does to p_1 in Figure 2.7). That is, m is sent before p has taken its local checkpoint, and received after p' has taken its local checkpoint. Further, let $c_1, \ldots, c_l$ be the route that m takes; in other words, $f(c_{k-1}, n_d) = c_k$ for $1 < k \leq l$. We show that m always travels along a channel c_i before m_{NS} can be sent on c_i.

Since allocation of a route is non-preemptive, and channels are FIFO, m will be put on channel c_1 before p can send out another message on channel c_1. In particular, m is put on c_1 before p can put m_{NS} on c_1.

We now show that m is put on all channels $c_1, \ldots, c_l$ before any processor can put m_{NS} there. This is done by induction. First, as noted above, m is put on c_1 before m_{NS} can be put there. Next, suppose that for all $k < l$, m travels

through c_k before a m_{NS} message can be put there. Before m_{NS} can be put there, m must be able to allocate c_{k+1}. Therefore m is put on channel c_{k+1} before m_{NS} is put on channel c_k. Now, since $f(c_k, n_d) = c_{k+1}$, $(c_k, c_{k+1}) \in E$, and m_{NS} will not be sent on c_{k+1} until after it has been received on c_k. As stated above, m_{NS} will not be received on c_k until after m has been put on c_{k+1}. Thus, m_{NS} will not be put on c_{k+1} until after m has been put there.

Therefore it is impossible for a cross-cut message to travel on a channel c after m_{NS} has been put there. Thus, when m_{NS} is received along a channel, it is guaranteed that no more cross-cut messages will be use that channel. □

Hypercube Interconnections

The Intel iPSC/2 and iPSC/860 [Nug88] are multicomputers with hypercube interconnection networks that are well-behaved. Each process is associated with a node on a hypercube. In a d-dimensional hypercube, nodes are numbered 0 through $2^d - 1$. If the binary representation of nodes i and j differ by exactly their k-th bit, then they are physically connected by a link that is said to be in dimension k. As an example, Figure 2.9 shows a 3-dimensional hypercube with its links.

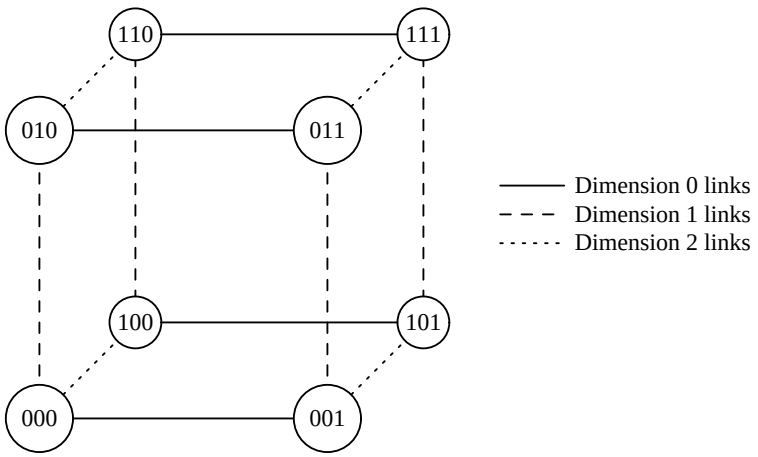

Figure 2.9: A 3-dimensional hypercube

Each link is bidirectional, and thus can be divided into two unidirectional channels. The iPSC/2 and iPSC/860 both use the *e-cube* routing function, which is deadlock-free. It specifies that the only channels used are those in the dimensions in that the binary representation of the source and destination nodes differ. These channels are used in increasing order of dimension. For example in Figure 2.9, a message from node 000 to node 111 will pass through the channels $(000, 001), (001, 011)$, then $(011, 111)$.

To put this in the notation of Dally and Seitz, let $c_i = (n_a, n_b)$. Then $f(c_i, n_d) = c_o$, where c_o is the channel emanating from n_b in the lowest dimension in that the binary representations of n_b and n_d differ. For example, in Figure 2.9, $f((000, 001), 111) = (001, 011)$.

The method for sending m_{NS} markers therefore works as follows:

- After processor p takes its local checkpoint, it sends m_{NS} on its outgoing channel c_0 of dimension 0. This is because c_0 always starts a message route: There are no c_j such that $(c_j, c_0) \in E$.
- P sends m_{NS} on its outgoing channel c_k of dimension k only after it has received m_{NS} on each incoming channel c_j of dimension $j < k$. This is because for each of those c_j, $(c_j, c_k) \in E$.

This method results in two m_{NS} markers sent per physical link. As each node has $\log n$ links incident to it, the total number of m_{NS} messages is $n \log n$, which is $O(l(n))$ for the hypercube.

Mesh Interconnections

As a second example for sending m_{NS} messages, we consider a toroidal mesh. The next generation multicomputers, including the Intel Delta and Paragon use toroidal mesh interconnection networks. In these interconnection networks, the nodes are laid out on a grid, and each node has two outgoing links: one to its right, and one going up. To be more formal, in a $R \times C$ toroidal mesh, $I = G(N, L)$, where:

$$N = \{n_{ij} \mid i < R \text{ and } j < C\}$$

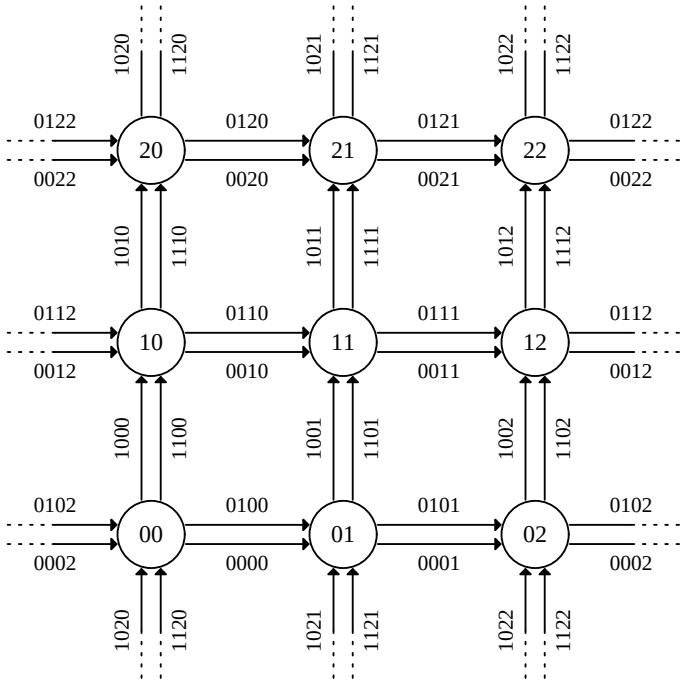

Figure 2.10: A 3×3 toroidal mesh

$$\begin{aligned} L &= \{(n_{ij}, n_{ij'}) \mid j' = (j+1) \bmod C\} \\ &\quad \cup \{(n_{ij}, n_{i'j}) \mid i' = (i+1) \bmod R\} \end{aligned}$$

As before, for the purposes of the deadlock-free routing function, we split each link into two channels. This time we call them *high* and *low* channels, for reasons discussed below. To be more clear, we give each channel a 4-digit identification number "$dhrc$," where

- d = channel's dimension: 1 if it is of the form $(n_{ij}, n_{i'j})$, 0 otherwise.
- h = 1 if it is a high channel, 0 if it is a low channel.
- r = row number of its source node.
- c = column number of its source node.

Figure 2.10 shows an example of a 3×3 toroidal mesh.

Deadlock-free routing on the mesh is similar to the routing on the hypercube. The message first gets routed to the correct column, and then to the correct row. When routing to the correct column, low channels are used when

the number of the current column is greater than the number of the desired column, and when routing to the correct row, low channels are used when the number of the current row is greater than the number of the desired row. We formally define f in terms of the current node n_{rc} and the destination node n_{ij} as follows:

$$f(n_{rc}, n_{ij}) = \begin{cases} c_{00rc} \text{ if } \ c > j \\ c_{01rc} \text{ if } \ c < j \\ c_{10rc} \text{ if } c = j \text{ and } r > j \\ c_{11rc} \text{ if } c = j \text{ and } r < j \end{cases}$$

The definition of f in terms of the current channel and destination node is slightly more complex, but follows directly. We omit it for the sake of clarity.

The first thing to notice about this function is that channels with identification numbers of the form c_{00r0}, $c_{01r(C-1)}$, c_{100c}, and $c_{11(R-1)c}$ never get used. Thus, they can be removed from the network. Next, a moment's thought shows that in terms of the channel identification numbers, this function is strictly increasing. In other words, if $(c_i, c_j) \in E$, then $i < j$. Therefore, the channel dependency graph has no cycles, and the routing function is deadlock-free.

The method for sending the m_{NS} messages works in a manner similar to the hypercube example: P_{ij} sends a m_{NS} message on outgoing channel c_{dhij} only after it has received m_{NS} messages from each incoming channel $c_{d'h'i'j'}$ where $d'h'i'j' < dhij$.

Thus, unlike the hypercube method, where every processor can start sending m_{NS} messages after checkpointing, on a mesh only the processors p_{i1} can start by sending m_{NS} on their channel c_{00i1}. Like the hypercube algorithm, however, it sends approximately two m_{NS} messages per physical link, which results in $4n$ total m_{NS} messages.

K-ary d-cubes and Express Cubes

A k-ary d-cube is a general class of interconnection network topologies in which there are k^d nodes. These nodes are laid out in d dimensions, with k sets of nodes in each dimension. Channels connect neighboring nodes that differ in

one dimension. Both hypercubes and toroidal meshes are examples of k-ary d-cubes: A d-dimensional hypercube is a binary d-cube, and a k-by-k toroidal mesh is a k-ary 2-cube.

General k-ary d-cubes are important as Dally has shown that multicomputers built using low-dimensional k-ary d-cubes for their interconnection networks give optimally low latency and high throughput [Dal90]. Accordingly, many new multicomputers, such as the Paragon, Stanford's DASH [LLG+90], the J-Machine [Dal91b], the Ametek 2010 [Sei88a] and the Mosaic [Sei88b], have been designed to be k-ary 2 and 3-cubes.

Since all of these machines use wormhole routing and are well-behaved, they too can be checkpointed with our Network Sweeping algorithm. We do not include the details here, but they follow as extensions of the mesh example above.

Finally, Dally has proposed *Express Cubes* as an optimization to standard k-ary d-cubes [Dal91a]. Express cubes use extra *express channels* for messages to jump over a number of nodes in each dimension so that the node-latency of a message's transmission is reduced. If the space allows, multiple express channels can be added hierarchically to each dimension to give more opportunity for greater throughput. He shows that express cubes both lower latency and increase throughput. Again, multicomputers built with express channels are well-behaved, and thus can be checkpointed with the Network Sweeping algorithm. The extension here is simply to have m_{NS} messages sent along express channels as well as the normal channels.

2.6.3 Optimizations

The Network Sweeping algorithm imposes an overhead of $O(l(n) + pp(n))$ extra messages. As stated above, $pp(n)$ is equal to the number of distinct pairs of processors that communicate during the checkpoint. In the worst case, this number is $O(n^2)$, but this worst case is only realized if every processor sends an application program message to every other processor in the multicomputer while the checkpoint is in progress (this is as opposed to, or rather is

a subset of Venkatesan's algorithm [Ven89], where the worst case is realized when every processor sends a message to every other processor during an entire checkpointing interval).

As a potential optimization (similar to one in Lai and Yang [LY87]), we can eliminate the need for extra m_{CL} markers if we simply attach an extra bit to each normal message in the system (on some multicomputers, like the iPSC/2, this bit comes for free, as there is an extra bit in message headers). Then, for messages coming from even checkpointing intervals, we set this bit to 0 and for those coming from odd intervals, we set it to 1. Now there is no need for m_{CL} messages after p_c's initial broadcast, because a processor can tell if it needs to checkpoint or if it needs to log an incoming message simply by looking at this extra bit.

Thus with this method, there is no possibility of degradation to $O(n^2)$ markers – instead, the number of extra messages is merely $O(l(n))$, which is $O(n \log n)$ for hypercubes and $O(n)$ for k-ary 2 and 3-cubes. This optimization does not come for free, however. In some multicomputers, there might not be any free bits in message headers. This combined with the fact that every receive operation must check the extra bit means that the regular processing of messages will be slowed down even in the absence of checkpointing. Thus, the degree of optimization (or degradation) depends on how much the multicomputer is slowed down by this extra bit, as well as on the frequency of checkpointing and the behavior of the program being checkpointed.

2.7 Recovery

Recovery is the same in all consistent checkpointing algorithms: Each processor reads its processor state from its local checkpoint, and messages from the message log are replayed. This replaying can take one of two forms: Either each processor can resend each of its messages that was logged by the receiving processor, or each processor can rebuild its queue of messages from its own message log. The latter form is superior since it involves no extra communi-

cation between processors, and no reading of other processors' message logs. This latter form is used in our multicomputer implementation.