PROGRAMMING A HYPERCUBE MULTICOMPUTER [+]

Sanjay Ranka, Youngju Won, and Sartaj Sahni
University of Minnesota

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

Abstract

We describe those features of distributed memory MIMD hypercube multicomputers that are necessary to obtain efficient programs. Several examples are developed. These illustrate the effectiveness of different programming strategies.

Keywords and phrases : Hypercube computers, MIMD computers, parallel computing

1 INTRODUCTION

Many applications such as weather forecasting, three dimensional modeling, fluid dynamics, computational chemistry, real time image processing etc. require computational capability far beyond what can be obtained from the fastest single processor computers available. There are essentially two ways in which this computational capability can be achieved. The first is to develop even faster single processor computers. The second is to use several computers in parallel to solve a single problem.

Until recently, much of the emphasis in high speed computing was in the first of these approaches. Recently, however computers supporting truly parallel computing have become commercially available. Hypercube computers are dominant in this class of commercially available parallel computers. Ametek, Floating Point Systems, Intel Scientific Computers, NCUBE and Thinking Machines are some of the vendors of hypercube and modified hypercube computers.

In Section 2, we describe the hypercube architecture. In Section 3, we develop an example to illustrate the nature of programming on a hypercube efficiently. Section 4 considers programming techniques specific to MIMD parallel computers.

2 HYPERCUBE ARCHITECTURE

Parallel computers may be classified by taking into account their memory organization, processor organization, and the number of instruction streams supported.

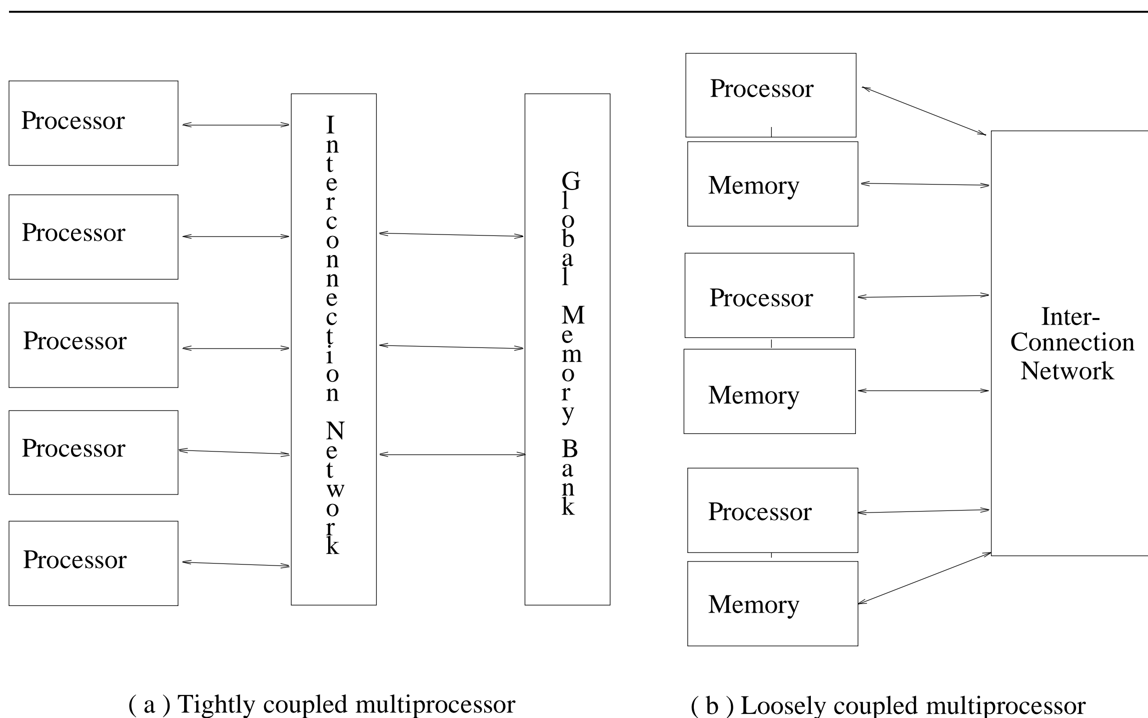

Figure 1 : Multiprocessor types

Memory organization

A *multiprocessor* is a parallel computer in which the (at least two) processors share a common memory or a common memory address space [QUIN87].

A block diagram of a *tightly coupled* multiprocessor is provided in Figure 1(a). In such a computer, the processors access memory via a processor-memory interconnection network. This

network could be a simple bus or any of a variety of switching networks such as Omega network, Benes network, full cross bar switch, etc [SIEG79] . In a *loosely coupled* multiprocessor, each processor has a local memory (Figure 1(b)). These local memories together form the shared address space of the computer. Typically a memory reference to the local memory of a processor is orders of magnitude faster than a memory reference to a remote memory as local memory references are not routed through the interconnection network while remote memory references are.

The block diagram for a *multicomputer* is shown in Figure 3.

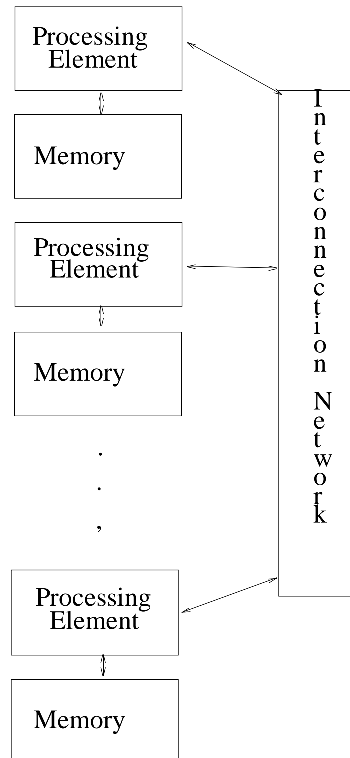

Figure 2 : A multicomputer

The significant difference between a multicomputer and a multiprocessor is that a multicomputer has neither a shared memory nor a shared memory space [QUIN87]. Consequently to use data in a remote memory, it is necessary to explicitly get that data into the local memory. This and all other inter-processor communication is done by passing messages (via the interconnection network) among the processors.

The distinction between multicomputers and multiprocessors is essentially that the former has no shared memory or address space while the latter has this. The NCUBE hypercube is a multicomputer. Our further discussion is restricted to multicomputers.

Processor organization

Processor organization is defined by the interconnection network used to connect the processors of the multicomputer. Some of the more common interconnection networks are: two dimensional mesh, ring, tree and hypercube (Figure 3). The first three are intuitive while the fourth needs some elaboration. In a hypercube of dimension d, there are 2^d processors. Assume that these are labeled $0, 1, \cdots, 2^d - 1$. Two processors i and j are directly connected iff the binary

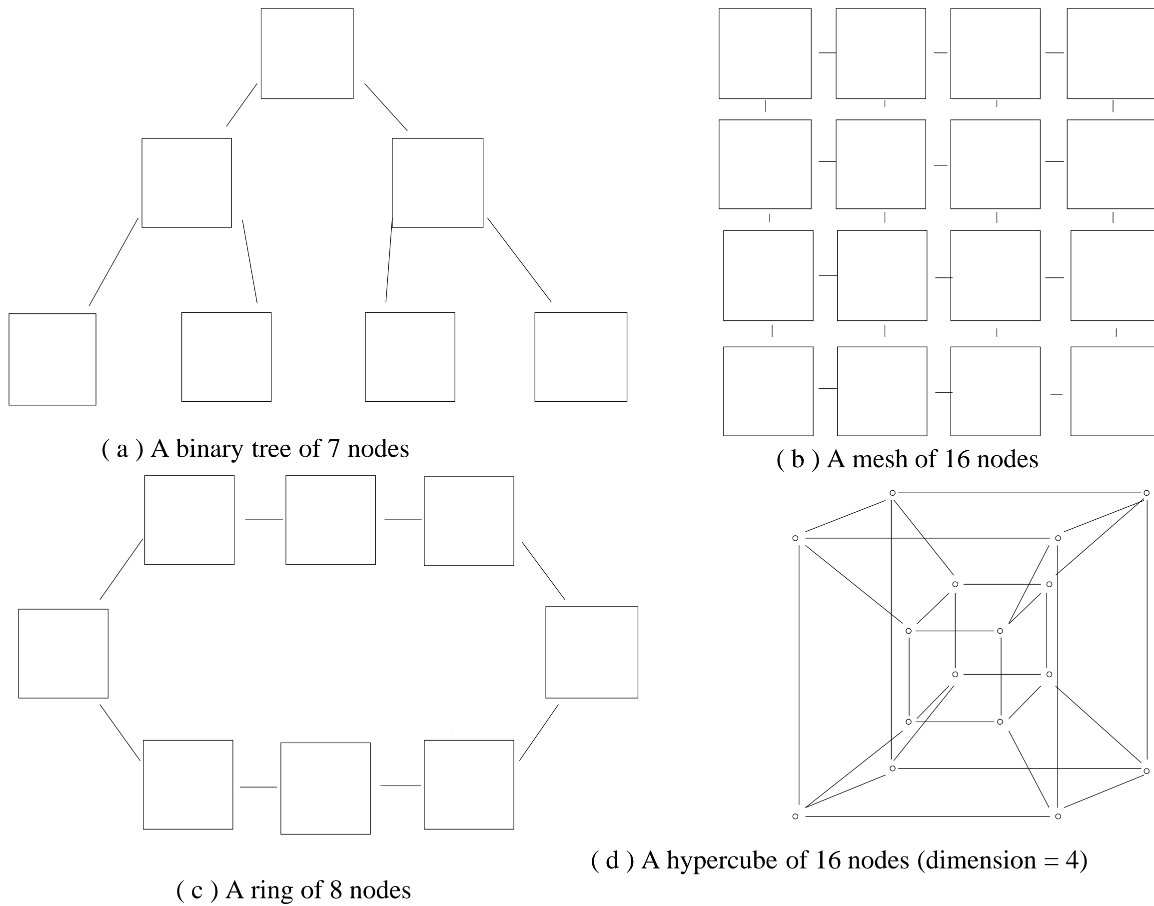

Figure 3 : Different types of interconnection network

representations of *i* and *j* differ in exactly one bit. Each edge of Figure 3(d) represents a direct connection. Thus in a hypercube of dimension *d*, each processor is connected to *d* others. If the direct connection between a pair of processors *i* and *j* is *unidirectional*, then at any given time messages can flow from either *i* to *j* or from *j* to *i*. In the case of *bidirectional* connections, it is possible for *i* to send a message to *j* and for *j* to simultaneously send one to *i*. The interconnections in NCUBE's hypercube are bidirectional.

The popularity of the hypercube network may be attributed to the following:

a) Using d connections per processor, 2^d processors may be interconnected such that the maximum distance between any two is *d*. While meshes, rings, and binary trees use a smaller number of connections per processor, the maximum distance between processors is larger. It is interesting to note that other networks such as the star graph [AKER87] do better than a hypercube in this regard. A star graph connects (d+1)! processors using d connections per processor. The inter-processor distance is at most $\left\lfloor \frac{3(d-1)}{2} \right\rfloor$. The hypercube has the advantage of being a well studied network while the star graph is relatively new.

b) Most other popular networks are easily mapped into a hypercube. For example a 2×4 mesh, 8 node ring, and a 7 node full binary tree may be mapped into an 8 node hypercube

as shown in Figure 4. A full binary tree cannot be mapped on the hypercube such that every two nodes of the tree are connected by a hypercube connection [DESH86]. In Figure 4(c) nodes 000 and node 110 have a connection via node 100. Node 100 is not used for any of the tree nodes. Thus, it only performs message passing between nodes 000 and 110. Gray codes are often used to obtain efficient mappings of meshes and rings onto a hypercube [CHAN86]. An i bit binary gray code S_i is defined recursively as below:

$$S_1 = 0,1;\quad S_k = 0[S_{k-1}],\ 1[S_{k-1}]^R$$

where $[S_{k-1}]^R$ is the reverse of the $k-1$ bit code S_{k-1} and $b[S]$ is obtained from S by prefixing b to each entry of S. So, $S_2 = 00, 01, 11, 10$ and $S_3 = 000, 001, 011, 010, 110, 111, 101, 100$. The ring to hypercube mapping of Figure 4.(a) uses S_3.

c) A hypercube is completely symmetric. Every processors' interconnection pattern is like every other processors'. Furthermore, a hypercube is completely decomposable into sub-hypercubes (i.e., hypercubes of smaller dimension).

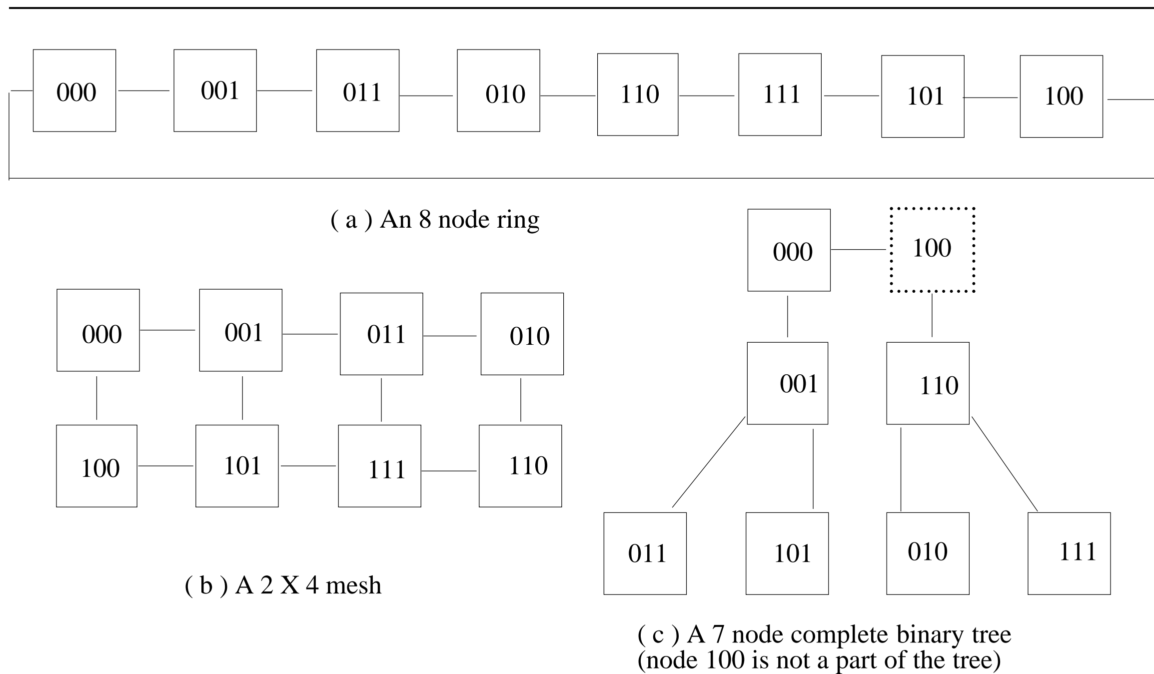

Figure 4 : Embeddings of different networks in a 8 node hypercube

Instruction streams

Flynn [FLYN66] classified computers based on the number of instruction and data streams. The two categories relevant to our discussion here are SIMD (single instruction multiple data streams) and MIMD (multiple instruction multiple data streams). In an SIMD parallel computer, all processors execute in a synchronous manner. In any given cycle, all processors execute the same instruction. MIMD parallel computers are generally asynchronous (in theory they could be synchronous too) and different processors may execute different instructions at any given time.

In this paper, we consider strategies to develop efficient programs for MIMD hypercube

multicomputers. When programming such a computer, one must be aware of the significant differences between the cost of arithmetics and that of communication in the commercially available machines. For instance, [DUNI86] has performed an experimental study of inter processor communication time and time to perform arithmetic operations on an NCUBE hypercube multicomputer. A summary is provided in Figure 5. >From Figure 5(a), we see that an 8 byte message transfer between two directly connected processors (i.e., processors 1 hop apart) takes 42 times the time for an 8 byte real addition and 32 times that of an 8 byte real multiplication. Furthermore, longer messages are transferred at a higher rate (i.e., bytes per second) than shorter ones going the same distance and it takes longer to send the same message to a processor 4 hops away than to one 2 hops away (Figure 5(b)). The time for a one hop communication of a message of length N bytes is approximately $446.7 + 2.4N$ microseconds [DUNI86].

Operation	Time	Comm./Comp.
8 byte transfer (1 hop echo)	470 µs	
8 byte real add.	11.2 µs	42
8 byte real mult.	14.7 µs	32

(a)

Communication speeds KB/s						
Length	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
8	17.2	11.7	8.9	7.1	5.9	5.1
16	33.1	22.4	16.9	13.7	11.4	9.8
32	61.6	41.7	31.4	25.2	21.1	18.1
64	106.6	72.1	54.4	43.7	36.5	31.1
128	169.5	114.4	86.1	69.1	57.6	49.4
256	241.2	162.2	121.4	97.1	81.0	69.5
512	304.8	203.4	152.4	121.9	101.6	87.1
1024	351.1	233.4	174.9	139.8	116.4	99.8
2048	380.8	252.2	188.8	150.8	125.6	107.6

(b)

Figure 5: Performance table

3 AN EXAMPLE PROGRAM

The template matching problem is a problem that is solved frequently in Computer Vision. We are given an N × N image I and an M × M template T. The *match* of the template to the image at point (i, j) is measured by the two dimensional convolution:

$$(1) \quad C2D(i, j) = \sum_{s=0}^{M-1} \sum_{t=0}^{M-1} I((i+s) \bmod N, (j+t) \bmod N) * T(s, t)$$

$O(N^2M^2)$ operations are required to compute C2D for all points (i,j) using (1) directly. In obtaining a hypercube program to compute the two dimensional convolution, we assume (for simplicity) that the hypercube is of dimension d, where d is even. Our discussion is easily extended to the case when d is odd. Further, we assume that the image matrix I is distributed

over the 2^d processors as indicated by the partitioning scheme of Figure 7. Here, processors are numbered using the gray code mesh mapping scheme of Figure 4. When our program is done computing the matrix C2D, C2D(i,j) will be in processor k iff I(i,j) was initially in this processor. Hence the partioning of C2D across the processors is identical to that of I.

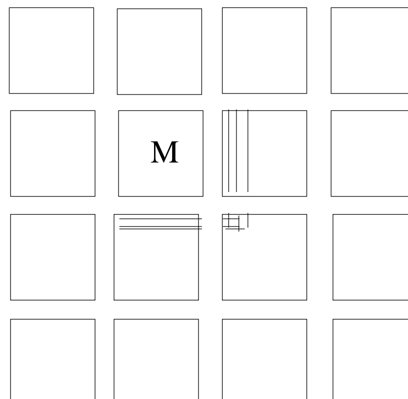

Figure 6: Region (of image values or result) required by M

Observe that no processor has all the image values needed to compute the C2D values in its partition (Figure 6). The additional image values required are with the east, south, and southeast neighbor processors (cf. mesh mapping). This difficulty may be overcome in one of two ways.

(1) The three neighbors cited above send the required image values to the processor which then does all the computing required for its C2D partition. The data to be transferred is shown in Figure 6 by vertical and horizontal lines.

(2) Each processor does all the computing involving the image values it has initially (this includes some of the computing for C2D values in its west, north, and northwest neighbor processors) and then transmits the partially computed C2D values to these neighbors. The partially computed values are shown in Figure 6 by horizontal and vertical lines.

If we assume that the size of the image values (in bytes) is the same as that of the C2D values, then both the schemes involve the same amount of data transfer. We continue with strategy (2). Two high level descriptions of the node program for each hypercube node are given in Figures 7 and 8. Notice that the latter program exploits the nonblocking nature of the node write (*nwrite*) function and attempts to overlap node computing and data transmission. Figure 9 is a high level description of the host program.

The run times of the C code of Figure 8 on an NCUBE hypercube are given in Figure 10. Since the amount of computation is much larger than the amount of communication, we did not find any significant differences between the run times of the programs of Figures 7 and 8. In the next section, we will develop an example in which the overlap of computation and communication leads to a significant reduction in run time.

```
int TemplateAtNode;
{
  if (nodeid == 0) Receive Template from host
  Broadcast Template from  node 0 (using tree expansion)
  Calculate Convolution for self
  Calculate Convolution for North node
  Calculate Convolution for West node
  Calculate Convolution for NorthWest node
  Send Convolution for NorthWest node
  Send Convolution for West node
  Send Convolution for North node
  for (i = 0; i < 3; i++)
  {
    Receive Convolution from a node
    Update Convolution
  }
  End Signal to node 0 (using tree collapse)
  if (nodeid == 0) Send End Signal to the host
}
```

Figure 7: Template Matching (High Level Description of the node)
No overlap between communication and computation

```
int TemplateAtNode;
{
  if (nodeid == 0) Receive Template from host
  Broadcast Template from  node 0 (using tree expansion)
  Calculate Convolution for NorthWest node
  Send Convolution for NorthWest node
  Calculate Convolution for West node
  Send Convolution for West node
  Calculate Convolution for North node
  Send Convolution for North node
  Calculate Convolution for self
  for (i = 0; i < 3; i++)
  {
    Receive Convolution from a node
    Update Convolution
  }
  End Signal to node 0 (using tree collapse)
  if (nodeid == 0) Send End Signal to the host
}
```

Figure 8: Template Matching (High Level Description of the node)
Overlap between communication and computation

4 PROGRAMMING CONSIDERATIONS

```
HostTemplate();
{
 Open a hypercube of the required dimension;
 Load the "node" program on all the nodes;
 Send the template to node 0;
 Receive Completion Signal from node 0;
 Deallocate the hypercube;
}
```

Figure 9: Template Matching (High Level Description of the host)

		m			
p	n	4	8	16	32
1	32	0.505	1.857	7.000	20.450
4	32	0.139	0.482	1.417	
	64	0.514	1.872	7.026	20.497
16	32	0.045	0.115		
	64	0.142	0.484	1.422	
	128	0.516	1.874	7.031	20.510
64	32	0.021			
	64	0.047	0.118		
	128	0.144	0.487	1.426	
	256	0.519	1.878	7.036	20.520

Times are in seconds

n = size of the image
m = size of the template
p = number of processors
Figure 10: Run times of the program in Figure 8

One view of programming is that it is the process of mapping an algorithmic abstraction onto a target computer. The resultant mapping (called a *program*) is specified in any one of the programming languages supported by the target computer. To arrive at an efficient program for a multicomputer, one needs to consider the following:

1) initial algorithmic abstraction - quite clearly, this has a significant influence on the resulting program.

2) mapping the abstraction on to the target computer . Here the following issues are important:

a) distributing the data across the local memories of the multicomputer

b) load balancing - ensuring that all the processors have a comparable computational load

c) repeating identical computations vs sharing results

d) overlapping computation and communication

e) number of processors to use

Before proceeding with a discussion of these considerations, it is appropriate to define two terms, speedup and processor utilization efficiency, that are commonly used when one talks about the efficiency of a multicomputer program.

Definition: Let t_0 be the time required to solve a problem using the fastest single processor program for that problem. Let t_k be the time required by the multicomputer program when k processors are in use. The *speed up*, S_k, obtained by the multicomputer program is:

$$S_k = \frac{t_0}{t_k}$$

The *efficiency of processor utilization*, E, is:

$$E = \frac{S}{k}$$

Barring any anomalous behavior, one strives for a speed up $S_k = k$ and $E = 1$. Anomalous behavior [LAI84] can result in $S_k > k$ and $E > 1$. In practice, because of the inter processor communication overhead, S_k will generally be less than k and E will be less than 1.

4.1 Algorithm Selection

There are essentially two approaches to obtaining the initial algorithm that is to be developed into a muticomputer program. The first is to start with an existing parallel algorithm and the second to develop a new algorithm. When the latter approach is used one is aware of the target architecture and is more likely to arrive at an algorithm that is efficiently and easily mapped into a program. When the first approach is used, one must keep the following in mind:

1) Is the required inter-processor communication pattern easily obtained on the target computer. For example, an algorithm requiring frequent and random exchange of messages will not perform well on a hypercube multicomputer. Furthermore, since the memory of a multicomputer is distributed across the multicomputer nodes, it must be possible to partition the data so that locality of memory reference is preserved to a large extent.

2) Let the *total work* done by an algorithm be the product of its run time and the number of processors used. For many of the asymptotically fastest parallel algorithms, this exceeds the total work done by the correspondingly fastest uniprocessor algorithms by more than a constant factor. So, for example, one can find the shortest paths between all pairs of vertices in an n vertex directed graph in $O(\log^2 n)^+$ time using $O(\frac{n^3}{\log n})$ processors, [DEKE81]. However, Floyd's dynamic programming algorithm [HORO85] runs in $O(n^3)$. Thus, the parallel algorithm of [DEKE81] does $O(\log n)$ times the work done by Floyd's algorithm. For the sake of concreteness, suppose that it does $4 \log n$ times the work. We can run the algorithm of [DEKE81] on a k processor hypercube by letting each processor of the hypercube do the work of $O(\frac{n^3}{k \log n})$ processors in the algorithm of [DEKE81]. However the resulting algorithm will not be faster than using Floyd's algorithm on a single processor unless the number, k, of hypercube processors exceeds $4 \log n$ (actually more are needed because of the reasons cited below). This means that if $n = 1024$, then we need at least $4\log_2 1024 = 40$ processors to break even with Floyd's algorithm on a single processor!. For larger n, more processors are needed just to catch up with the uniprocessor algorithm!. $^+$ all logarithms are assumed to have a base of 2 Floyd's algorithm can, however, be parallelized to run in $O(\frac{n^3}{k})$ time for $k \le n^2$ (k is the number of processors), [JENQ87]. While this does not yield the asymptotically fastest algorithm, it can be mapped into a multicomputer program that exhibits acceptable speedups. To be effective, the total work done by the initial parallel algorithm must be within a constant factor of the work done by the fastest uniprocessor

algorithm (unless the number of processors available is very large).

3) Even when the communication pattern is suited to the target architecture and the total work done is not excessive relative to the work done by the best uniprocessor algorithm and the number of processor available, the algorithm may not result in a satisfactory program. This is so as most of the existing parallel algorithms with good asymptotic behavior assume that the cost of inter-processor communication is comparable to that of a basic operation (i.e, an add, subtract, etc.). As a result, these algorithms do not attempt to reduce communication at the expense of arithmetics. However, on commercially available multicomputers such as NCUBE/10, communication is significantly more expensive than basic arithmetic operations (cf. Figure 5).

4.2 Data distribution

In a multicomputer, it is necessary to distribute the data across the local memories for the following reasons:

1) There may be more data than can be accommodated in the local memory of a single node processor.

2) Local memory access is much faster than access to a remote memory, hence each processor should have as much of the data it needs in its own local memory.

In some cases, it is possible to reduce or even eliminate inter-processor communication by replicating some or all of the data across some or all of the processors. For example, consider the template matching problem of the previous section with the following assumptions:

a) The image and template matrices are initially in the host

b) The convolution matrix is to be left on the hypercube after it has been computed. This is to be partitioned across the processors as in the previous section.

One way to accomplish this task is to distribute the image and template matrices to obtain the distribution required by the program of Figure 8. Another possibility is to distribute the image matrix so that each processor has all the image values it needs to compute the convolution at all the points assigned to it. Thus, in addition to the data a processor begins with in Figure 8 , it has the nece ssary data that was previously only in its east, south-east and south neighbor processors (shown by horizontal and vertical lines in Figure 15). In this case, the hypercube processors do not need to communicate any image or partial convolution values to their north, west, and northwest neighbors. When this image distribution is used, the node program is simpler. Furthermore, the computation time is reduced. Figure 11 gives a high level description of the new node program. Figure 12 gives the times for the two different cases. These times include the time needed to transmit the image and template matrices from the host to the hypercube . Hence the times of the program of Figure 8 are higher in the table of Figure 12 as compared with Figure 10.

While data replication may be desirable, the size of the local memory will often limit the extent to which it is possible. As noted in [JENQ87], the all pairs shortest path problem is efficiently solved using the single source all destinations algorithm of Dijkstra [HORO85]. However, this requires the full cost matrix to be present in the local memory of each processor. For large graphs, this is not possible. Hence, it is necessary to use other algorithms.

4.3 Load Balancing

The objective of load balancing is to obtain an approximately equal distribution of the work load across the multicomputer processors. When the work load is known *a priori*, it is generally the responsibility of the host to ensure such a work load distribution. When the work load is not known *a priori*, the multicomputer nodes need to dynamically readjust the load. We shall concern ourselves only with this latter case of dynamic load balancing.

Figures 13 and 14 give two variations of the same heuristic to balance the load. In both, load balancing is accomplished by averaging over the load in processors that are directly

```
TemplateAtNode; {
  if (nodeid == 0) receive template from host
  Broadcast Template from node 0 (using tree expansion)
  Receive Image from the Host
  Perform Convolution;
  End Signal to node 0 (using tree collapse)
  if (nodeid == 0) Send End Signal to the host }
```

Figure 11: Template Matching (High Level Description of the node program)
The node has all the necessary image values for convolution

	Figure 8			Figure 11		
p	m			m		
	4	8	16	4	8	16
16	0.902	2.422	8.143	0.760	1.731	5.413
64	0.637	1.176	2.532	0.607	1.016	2.345

Times are in seconds

Size of the image = 128×128
m = size of the template
p = number of processors
Figure 12: Table comparing performance of programs in Figure 8
and Figure 11.

```
LoadBalance1();
{
 for(i = 0; i < CubeSize ; i++) {
   Send MyLoad to neighbor processor along dimension i;
   Receive HisLoad from neighbor  processor along dimension i;
   and append to Myload;
   avg =  (MyloadSize + HisLoadSize + 1) / 2;
   if (MyLoadSize > Avg) MyloadSize = Avg;
      else if ( HisLoadSize > Avg) MyLoadSize += HisloadSize - Avg;
  }
}
```

Figure 13: Load balancing (Heuristic 1)

connected. The variables used have the following significance:

MyLoad = current load in the node processor
HisLoad = load in a directly connected node processor

MyLoadSize = size of the load in the node processor
HisLoadSize = size of the load in a directly connected node processor

```
LoadBalance2();
{
 for(i = 0; i < CubeSize ; i++) {
   Send MyLoadSize to neighbor processor along dimension i;
   Receive HisLoadSize from neighbor processor along dimension i;
   avg =  (MyloadSize + HisLoadSize + 1) / 2;
   If (MyLoadSize > Avg) {
      Send extra load (MyLoadSize - Avg) to neighbor processor along
      dimension i;
      MyloadSize = Avg;
     }
     else if ( HisLoadSize > Avg) {
      Receive  extra load (Avg - HisLoadSize) from neighbor processor
      along dimension i;
      MyLoadSize += HisloadSize - Avg;

  }
}
```

Figure 14: Load balancing (Heuristic 2)

avg = average size of the load of the two processors

The only difference between the two variations is that in the first one a processor transmits its entire work load (including the necessary data) to its neighbor processor while in the second variation only the amount in excess of the average is transmitted. However, in order to achieve this reduction in load transmission, it is necessar y to first determine how much of the load is to be transmitted. This requires an initial exchange of the load size. Hence variation 2 requires twice as many message transmissions. Each message of variation 2 is potentially shorter than each message transmitted by variation 1. We expect variation 1 to be faster than variation 2 when the number of bytes in MyLoad and HisLoad is relatively small and the time to set up a data transmission relatively large. Otherwise, variation 2 is expected to require less time.

Before incorporating a load balancing scheme into an algorithm, it is necessary to weigh the potential reduction in time required to complete the work against the time required to balance the node. If the latter is larger, our algorithm will perform bett er when the load is not dynamically balanced by us. [RANK88] reports on the effect of introducing a load balancing step into the computation of the Hough Transform. This resulted in a significant reduction in run times.

4.4 Replication of Computation

When computing on a conventional uniprocessor computer, we can reduce the run time by repeatedly using a computed value rather than recomputing the value each time it is needed. Thus, the code of Figure 15(a) runs faster than that of Figure 15(b). On a multicomputer with 16 processors, it would be faster to have the i'th processor compute $f_i(g(x))$, $1 \le i \le 16$ (assuming all 16 have the value of x) rather than to have to have one compute $g(x)$ and then broadcast the value to the remaining 15 processors. Recomputing $g(x)$ (Figure 15(d)) will be faster than reusing $g(x)$ (Figure 15(c)) by approximately the time needed to broadcast $g(x)$.

$y = g(x)$
do 100 $i = 1$, 16
100 $z_i = f_i(y)$

(a) $g(x)$ computed only once

do 100 $i = 1$, 16
100 $z_i = f_i(g(x))$

(b) $g(x)$ computed 16 times

if (ProcessorId = 0)
then $y = g(x)$
Broadcast y to other processors
else Receive $g(x)$ in variable y
Compute $f_i(y)$

(c) Only processor 0 computes $g(x)$

$y = g(x)$
Compute $f_i(y)$

(d) Each processor computes $g(x)$

Figure 15

4.5 Overlapping Computation and Communication

In Section 4, we saw an example where rearranging the computation so as to overlap computation and communication resulted in an improvement in program performance. Since this is a very important aspect of multicomputer programming, we develop another example in this section. This example has to do with finding a shortest path in a maze.

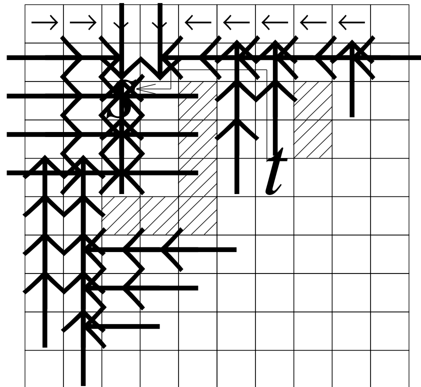

Figure 16 : Shortest path in a maze

A shortest path between s and t in the maze at Figure 16 can be found using a breadth first search. Shaded cells are blocked and the path cannot go through them. First, cells that are one unit from s are labeled, then those 2 units from s are labeled, then those that are 3 units from s are labeled, and so on. This labeling continues until the target cell t is reached. Blocked cells are not labeled. Four labels: $\rightarrow$, $\leftarrow$, $\downarrow$, and $\uparrow$ are used to point to the cell from which we reached the

current cell. Now the shortest path can be identified simply by following arrows from t to s. Cells which are the same distance from cell s are called front wave cells and thus the labeling process is often called *front wave expansion*. This *front wave expansion* has been implemented on an NCUBE by Won and Sahni [WON87], and two high level algorithms are given in Figures 17 and 18. The former is a straightforward implementation while the latter overlaps computation and communication. A possible maze partitioning is given in Figure 19.

Step 1 **[Maze partitioning and mapping]** Partition the $n \times n$ maze into k parts and assign one partition to each of the k node processors.

Step 2 **[Front wave expansion]** Each processor that has a maze cell on the current front wave expands the front wave. This expansion may require communicating with other processors as the cells adjacent to the front wave cell being expanded may be in different processors. All communication requests are saved.

Step 3 **[Inter processor communication]** Each processor sends its communication packets to the destination processor.

Step 4 **[Process communication packets]** Each processor examines the packets it receives and labels the front wave cells contained in these packets.

Step 5 Repeat steps 2, 3, and 4 until either the target cell is reached or the new front wave has no cells in it.

Figure 17 : Algorithm 1 for front wave expansion

Step 1 **[Maze partitioning and mapping]** Partition the $n \times n$ maze into k parts and assign one partition to each of the k node processors.

Step 2 **[Inter processor communication]** Each processor sends its communication packets to the destination processor (front waves of distance d)

Step 3 **[Front wave expansion]** Each processor that has a maze cell on the current front wave expands the front wave (of distance d). This expansion may require communicating with other processors as the cells adjacent to the front wave cell being expanded may be in different processors. All communication requests are saved for the next iteration.

Step 4 **[Process communication packets]** Each processor examines the packets it receives and labels and expands (as in step 3) the distance d front wave cells contained in these packets.

Step 5 Repeat steps 2, 3, and 4 until either the target cell is reached or the new front wave has no cells in it.

Figure 18 : Algorithm 2 for front wave expansion

To faciliate the *front wave expansion*, each processor maintains a queue of front wave cells that are in its maze partition. During front wave expansion, each cell on this queue is expanded (i.e., the cells to its north, south, east, and west on the routing grid are examined). Some of these cells are in the processor's grid partition while others are in the grid partitions assigned to other

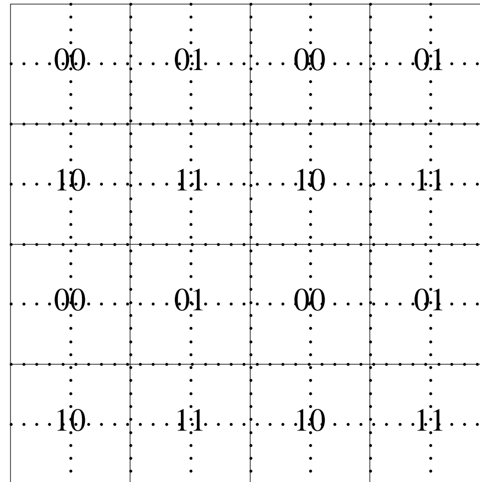

Figure 19 : Maze partitioning

processors. For those that are in the local partition, we may complete the front wave expansion. I.e., unblocked cells are labeled and put on an internal queue for later expansion. Cells that are not in the local partition are stored in a send queue for later transmission to the proper processors. Once the front wave cells have been examined in this way, each node processor transmits the cells in its send queue to the processors assigned to them. These are received by the destination processors and stored in their receive queues. The cells received (i.e., those in the receive queues) are processed. This involves labeling them and adding them to the internal queue if they are unlabeled and unblocked.

In algorithm 1 (Figure 17), following the sending of data packets in Step 3, a processor is ready to do further work. However the next step causes it to wait as there is a delay between data leaving a source processor and arriving at its destination. We can rearrange the computation to obtain algorithm 2 (Figure 18). This introduces a step between the send data and receive data steps. In the first iteration, each processor sends distance 0 (i.e., null packets) front wave cells in Step 2; processes distance 0 front wave cells in Step 3; and receives and processes remaining distance 0 packets in Step 4. In the next iteration, distance 1 packets are sent in Step 2; the local distance 1 front wave cells are processed in Step 3; and the remaining distance 1 front wave cells are received from the neighbor processors and processed in Step 4; and so on.

On the tested mazes, algorithm 2 required 25% to 30% less time than algorithm 1. This underscores the importance of reducing the communication overhead by overlapping communication with computation.

4.6 Number of Processors

For any given instance of a problem, there is an optimal hypercube dimension d to use. Using a hypercube of larger or smaller size will result in an increase in the program run time. While it is easy to expect an increase in run time using a smaller dimension hypercube, it is less evident why this should happen with a larger dimension hypercube. Several of the factors that contribute to this seemingly anomalous phenomena are:

1) The host to node data distribution time increases as more message transfers need to be set up.
2) Total inter processor communication time may increase.
3) In several applications the use of a multiple processors results in a recombination phase where the partial results computed by the individual processors is combined to obtain the overall solution. For example, consider the 0/1-Knapsack problem:

$$\textit{maximize} \sum_{i=1}^{m} p_i x_i$$

$$\textit{subject to} \sum_{i=1}^{m} w_i x_i \le c$$

$$\text{and } x_i \in \left\{0, 1\right\}, 1 \le i \le m$$

This problem was solved on an NCUBE by Lee et al. [LEE87] by first partitioning an m object instance into smaller instances, solving the smaller instances using dynamic programming on each hypercube node, and combining the results from these. Figure 20 shows the total time, time required to solve the smaller instances using dynamic programming, and the combining time on a test set with $m = 100$ and $c = 30$. The combining time increases as the number of processors increases. This increase eventually overshadows the reduction in the dynamic programming time.

Good speed up will be observed on larger hypercubes only when one is solving sufficiently large problem instances.

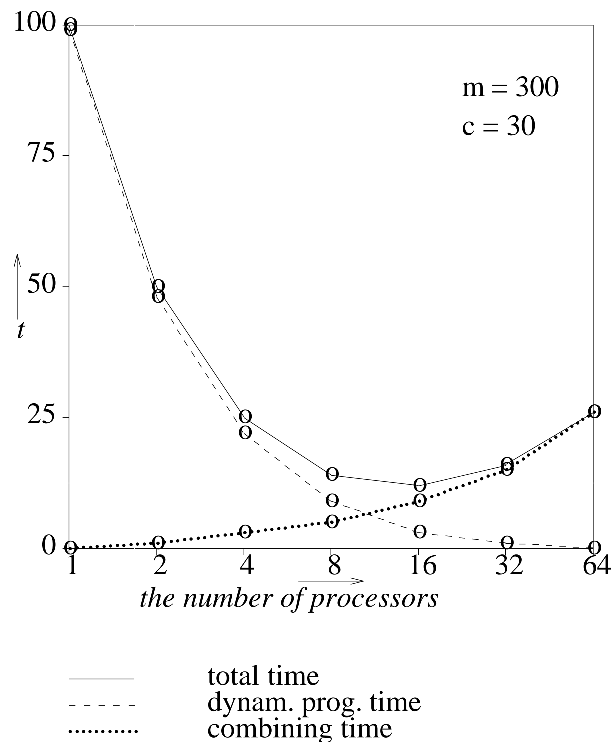

Figure 20 : Components of elapsed time for Knapsack Problem

5 CONCLUSIONS

The commercial availability of low cost multicomputers has opened new avenues of Computer Science research - operating systems, languages, software development environments, algorithms, etc. This paper has examined several issues that arise in programming a hypercube multicomputer. To achieve the full promise of these computers, it is necessary to pay careful attention to these issues.

6 REFERENCES

[AKER87]S. B. Akers, D. Harel and B. Krishnamurthy, "The Star Graph: An attractive alternative to the n-Cube", *Proc. of Intl. Conference on Parallel Processing*, **1987**.

[CHAN86]T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor", *IEEE Transactions on Computers*, **vol. C-35**, **Nov. 1986**, pp.

[DEKE86]E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM Journal on computing,* **1981**, pp. 657-675.

[DESH86]S. R. Deshpande and R. M. Jenevein, "Scalability of a binary tree on a hypercube", *Proceedings of the 1986 Intl. Conf. on Parallel Processing,* **1986**, pp. 661-668.

[DUNI86]T. H. Dunigan, "Hypercube performance", *Proceedings of 2nd Conf. on Hypercube multiprocessors, Knoxville,* **1986**, pp. 178-192 .

[FLYN66]M. J. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE,* **54** , **Dec. 1966**, pp. 1901-1909.

[HORO85]E. Horowitz and S. Sahni, "Fundamentals of Data Structures in Pascal", *Computer Science Press,* **1985**.

[JENQ87]J. Jenq and S. Sahni, " All pairs shortest paths on a hypercube multiprocessor", *Proceedings of Intl. Conf. on Parallel Processing,* **Aug. 1987**.

[LAI84]T. H. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms", *Communications of the ACM* **27**, **June 84**, pp. 594-602.

[LEE87]J. Lee and S. Sahni, "0/1 Knapsack problem on a hypercube multiprocessor system", *Proceedings of Intl. Conference on Parallel Processing,* **Aug. 1987**.

[QUIN87]M. J. Quinn, "Designing efficient algorithms for parallel computers", *McGraw-Hill Inc. ,* **1987**.

[RANK88]S. Ranka and S. Sahni, "Hough's transform on a hypercube multiprocessor computer", *University of Minnesota Technical Report*, In preparation.

[SAHN85]S. Sahni, "Software development in Pascal", *Camelot Publishing Company*, **1985**.

[SIEG79]H. J. Siegel, "Interconnection networks for SIMD machines", *IEEE Trans. on Computers,* **12**, **June 1979**, pp. 57-65.

[WON87]Y. Won and S. Sahni, "Maze routing on a hypercube multiprocessor computer", *Proceedings of Intl. Conf. on Parallel Processing*, **Aug. 1987**.