PARALLEL COMPUTING ON A HYPERCUBE: AN OVERVIEW OF THE ARCHITECTURE AND SOME APPLICATIONS*

George Ostrouchov
Mathematical Sciences Section
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

ABSTRACT

A hypercube parallel computer is a network of processors, each with only local memory, whose activities are coordinated by messages the processors send between themselves. The interconnection network corresponds to the edges of an n-dimensional cube with a processor at each vertex. This paper gives an overview of the hypercube architecture and its relation to other distributed-memory message-passing multiprocessors.

The computation of a crossproducts matrix is an important part of many applications in statistics and provides a simple yet interesting example of a hypercube algorithm. This example is presented to illustrate the concepts in programming a hypercube.

*Research sponsored by the applied Mathematical Sciences Research Program, Office of Energy Research, U. S. Department of Energy under contract DE-AC05-84OR21400 with the Martin Marietta Energy Systems, Inc.

Parallel Computing on a Hypercube: An Overview of the Architecture and Some Applications

George Ostrouchov, Oak Ridge National Laboratory

Abstract. A hypercube parallel computer is a network of processors, each with only local memory, whose activities are coordinated by messages the processors send between themselves. The interconnection network corresponds to the edges of an n-dimensional cube with a processor at each vertex. This paper gives an overview of the hypercube architecture and its relation to other distributed-memory message-passing multiprocessors.

The computation of a crossproducts matrix is an important part of many applications in statistics and provides a simple yet interesting example of a hypercube algorithm. This example is presented to illustrate the concepts in programming a hypercube.

1. Introduction. Interest in parallel computing has been growing for many years because of its potential for very high performance. Hypercube designs were discussed already in the 60's and 70's (for example [11]), but building such machines was impractical due to their complexity and the large number of components they would require. It was only after recent advances in VLSI technology that hypercube parallel computers became practical. Researchers at Caltech completed the first hypercube parallel computer (the 64-processor Cosmic Cube) in 1983 [10] and since then have successfully applied it and its successors to numerous scientific applications [5].

Commercial hypercubes started to appear in 1985 and currently are available from at least four vendors. These are the Intel iPSC with up to 128 processors, the Ametek's System/14 with up to 256 processors, the NCUBE Corporation's NCUBE/ntx-ten with up to 1024 processors, and Floating Point Systems T Series with up to 16384 processors (The FPS design makes a 16K-processor machine possible, but large configurations have yet to be built.). Hypercube overviews specific to some of these machines are given in [7] and [8]. A performance comparison of three hypercubes is given in [2]. At ORNL we have the Intel and NCUBE machines, so that most of our experience is based on these two, however this overview is intended to be more general. The hypercube is now perhaps the most successful large-scale parallel architecture and the currently available machines have the potential of supercomputer or near supercomputer performance at a relatively low cost.

The hypercube parallel architecture falls into the MIMD (Multiple Instruction Multiple Data) category of Flynn's taxonomy [3]. From a user point of view, there are two broad classes of MIMD multiprocessors that can be defined by the placement of memory and mode of communication. In a *shared-memory* multiprocessor, the processors communicate by accessing a memory or a set of memories common to all processors over a switching network. In a *distributed-memory* multiprocessor, each processor has only local memory, and the processors communicate by sending messages between themselves over a communication network. Of course, systems with both local and shared memory are possible. The distinction between the two broad classes is not always clear cut at the hardware level and it is the programming environment that is either shared-memory or distributed-memory message-passing. The hypercube is considered a distributed-memory message-passing multiprocessor. The communication network between processors is perhaps the most critical issue in the design of distributed-memory multiprocessors. A further breakdown of distributed-memory message-passing multiprocessors is thus based on the communication network topology and one of these is the hypercube topology.

The remainder of this paper is organized as follows. Section 2 describes the hypercube architecture in detail and shows how other architectures can be embedded in the hypercube. The programming environment is presented in § 3 and an example application and its implementation are given in § 4.

2. The Hypercube Architecture. A hypercube parallel computer is a network of processors, each with only local memory, whose activities are coordinated by

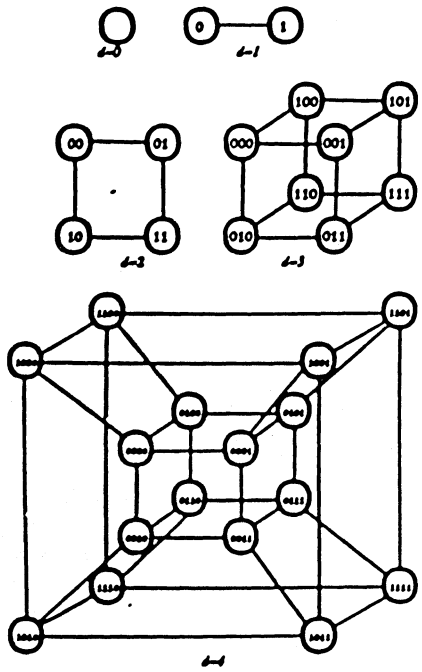

FIG. 1. *Hypercubes of dimension zero through four.*

messages the processors send between themselves. A d-dimensional hypercube has 2^d processors. The communication network corresponds to the edges of a d-dimensional cube, where each vertex or node is a processor. Each node is uniquely identified by d binary digits and the binary tags of any two neighbors differ by exactly one bit. Thus to communicate with each of its neighbors, each node has degree d (has d communication channels). To illustrate the interconnection scheme, consider the hypercubes of dimension 0 through 4 in Fig. 1.

A d-dimensional hypercube is obtained by duplicating a $(d-1)$-dimensional hypercube and connecting corresponding nodes. The binary tags are prefixed by a 0 in the original hypercube and by a 1 in the duplicate to obtain the new binary tags. Note that each bit in the tags is associated with a dimension of the hypercube. Flipping a given bit in the tags defines pairs of neighbors across a given dimension. For example, flipping the middle bit in the d-3 hypercube of Fig. 1 defines neighbors in the vertical dimension.

Communication between neighbors is sent directly over the communication link between them. If a message needs to be sent between two nodes that are not neighbors, the message is routed through intermediate nodes until it reaches its destination. The routing algorithm is very simple. At each stage, the message is sent to a neighbor whose binary tag is one bit closer to the destination binary tag. The path length is thus the number of bit positions in which binary tags of the two nodes differ. Several routes are possible depending on the order in which the binary tag digits are considered. Often a communication coprocessor is also present on each node to free each main node processor of most of the communication overhead. In fact, messages that are only passing through may be processed entirely by the coprocessor.

The hypercube architecture has a good balance between node degree (number of channels per node) and diameter (maximum path length between any two nodes). Ideally, an interconnection scheme should have a small diameter for fast communication and nodes should have small degree, so that the scheme easily scales up to a large number of processors. Table 1 gives these two parameters for a number of interconnection schemes, including the hypercube, for n processors. Both the tree and the hypercube have a good balance in terms of these two parameters. However, the tree is not a homogeneous structure and a communication bottleneck will occur at the root in many applications, because only a single path exists between any pair of nodes. In a hypercube, on the other hand, there are d disjoint paths between any pair of nodes. This can be exploited to enhance communication bandwidth and fault tolerance.

Many interconnection schemes can be embedded in a hypercube. These include the ring, the 1-d, 2-d and 3-d mesh, as well as trees. Fig. 2 illustrates some of these interconnection schemes and Fig. 3 illustrates their embeddings in a hypercube. Most regular embeddings are accomplished by the use of Gray codes. These are sequences of d-bit strings such that successive strings differ by a single bit. Binary tags of neighbor nodes in a hypercube differ by a single bit, so Gray codes define paths in hypercubes. Those communication patterns that cannot be embedded in a hypercube, such as a complete graph, will still execute efficiently, because the longest path (*diameter*) in a hypercube with 2^d processors is only d and the average path length is $d/2$. For these

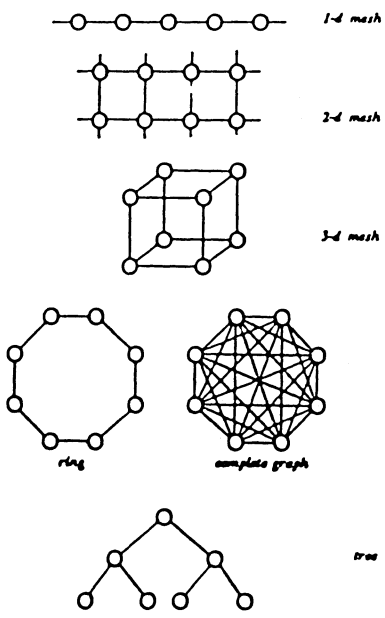

FIG. 2. *Some other interconnection schemes.*

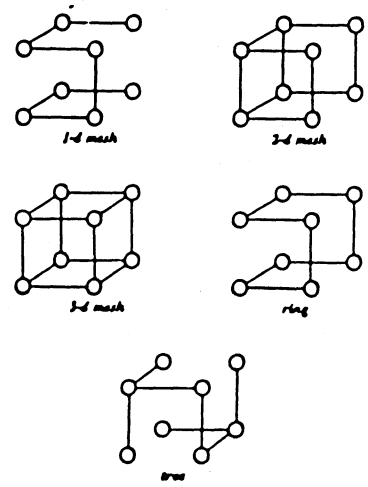

FIG. 3. *3-d hypercube embeddings of some schemes in Fig. 2.*

some schemes in Fig. 2.

Table 1

Interconnection	Diameter	Degree
1–d mesh	$n-1$	2
2–d mesh	$2(n^{1/2}-1)$	4
3–d mesh	$3(n^{1/3}-1)$	6
ring	$\lfloor n/2 \rfloor$	2
binary tree	$2(\log_2 n-1)$	3
hypercube	$\log_2 n$	$\log_2 n$
complete graph	1	$n-1$

reasons the hypercube is suitable for a very broad class of applications and is an excellent test bed for parallel algorithms. Many standard embeddings and communication patterns have been developed for the hypercube and are already available in the form of algorithms or subroutines [4], [9].

The processors at all nodes are usually identical and the hypercube looks the same from any node. Despite the attraction of this homogeneity, it is useful to have a separate processor in addition to the hypercube nodes that acts as a manager. This processor is usually called the host and has communication links with all or some subset of the hypercube nodes. Its management functions are performed loosely by sending messages over its communication links.

2. The Programming Environment. Each node of the hypercube has an operating system kernel that provides run-time support for user processes running on the node and handles communication with other nodes and the host. The host computer has an operating system, for program development and run-time support, as well as communication with the hypercube. User programs are developed and compiled on the host, and then the object code is loaded onto the node processors. The computation is initiated by the host and proceeds asynchronously on each node. Coordination is achieved by the exchange of messages that contain data or control information.

Since computation proceeds sequentially on each node and only local memory is available, no programming language extensions beyond those for communication and node identification are needed. A minimal set of extensions would include *send* and *receive* for communication and *whoami* for node identification. The extensions are different on each machine, but all include at least one or more versions of these three.

The programming languages used are usually C and FORTRAN. The exception is the FPS T Series using the British language Occam, which combines operating system and programming capabilities. However, a UNIX operating system is being developed even for this machine, at the cost of some overhead and slightly less flexibility, so that C and FORTRAN will likely be available soon. Despite the differences between the various hypercube implementations, it is possible to talk about program portability. If we define a generic *send*, *receive*, and *whoami* and write the application code with these generic calls, portability can be achieved by writing a set of machine specific communication subroutines for each machine. A number of programs written at ORNL are portable between the IPSC and NCUBE machines.

The most popular method of programming a hypercube is to write a single program that will run asynchronously on each node working on different data. Some applications can be split into homogeneous sections that naturally result into an identical program on each node. When this is not the case, it is still easier to write a single program whose function will differ depending on the node where it is running rather than 64 or 128 different programs. The best way to illustrate this is by an example, which is given in the next section.

Debugging facilities are improving, but still have a long way to go. For example, processor lights on the Intel iPSC indicate if processors are busy, idle, or communicating. This is an ironic throwback to the early days of computing, when it was useful to watch computer console lights to tell what was going on. Similar information can now be more conveniently obtained from system-log post-processors. For example SIEPCUBE, developed at Tufts U., gives a color graphics slow-motion replay of a parallel program execution from the system log produced during execution. Among other information, it displays communication patterns and busy-idle states of processors and at the time it works on the Intel or NCUBE hypercubes. Such displays are useful not only in debugging but also for improving efficiency of parallel programs. Much current research in computer science is concentrating on programming aids for parallel processing. Some real-time debuggers have already been developed and will likely become available soon. Hypercube simulators (for example [1]), however, are still the most used debugging tools, because they run in familiar sequential environments and allow the use of simple debugging techniques such as inserting print statements.

Programming a hypercube is becoming easier because of several factors. First, many bugs present in early versions of operating systems and compilers have been corrected and more efficient versions were developed. Second, more debugging aids are becoming available. And third, libraries of communication subroutines, matrix operations and factorizations, and other useful subroutines are being developed and some are already available. Also, an ever increasing number of researchers, particularly within computer science, are becoming familiar with programming hypercube multiprocessors. Thus the researcher in statistical computing has better chances of finding a knowledgeable source of help.

4. An Example. The computation of a crossproducts matrix is an important part of many methods in statistics and provides a simple yet interesting implementation on a hypercube. Suppose we have an $n \times p$ matrix X and we wish to compute the $p \times p$ crossproducts matrix $A = X'X$. If γ is the time for a multiply and an add operation, then the computation of A requires approximately $\frac{1}{2}np^2\gamma$ time on a single processor.

Note that the computation of A can be divided into a sum of r crossproduct computations by partitioning X into r blocks of rows

$$X = \begin{vmatrix} X_1 \\ X_2 \\ \vdots \\ X_r \end{vmatrix}$$

and computing $A = \sum A_i$, where $A_i = X_i'X_i$. Given a

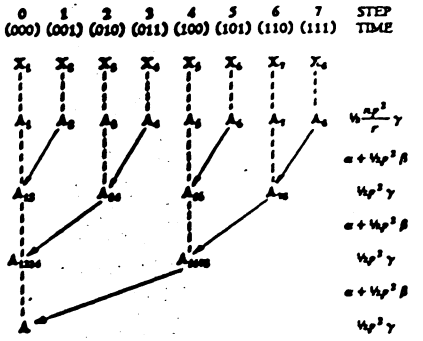

FIG. 4. *Computation of* $A = \sum X$ *on an 8-processor hypercube, with final result on processor 0.*

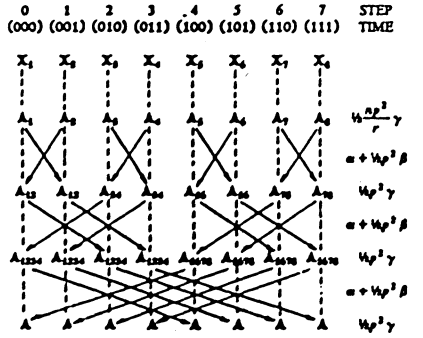

FIG. 6. *Computation of* $A = \sum X$ *on an 8-processor hypercube, with final result on all processors.*

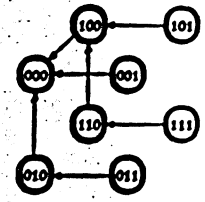

FIG. 5. *Communication pattern for Fig. 4.*

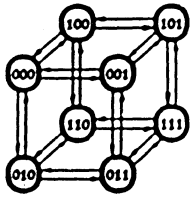

FIG. 7. *Communication pattern for Fig. 6.*

d-dimensional hypercube with $r = 2^d$ processors, we can compute all A_i's concurrently. This takes approximately $\frac{1}{2}\frac{np^2}{r}\gamma$ time. We have thus reduced the time r fold, but we are not done because the A_i's need to be summed.

The fastest way to compute a sum in parallel is by pairwise summation. Incidentally, pairwise summation has better numerical properties than simple sequential summation. Fig. 4 illustrates how this may be done on a $d = 3$ hypercube. Each column lists the matrices computed by a given processor and arrows indicate where a partial sum is communicated to another processor. If no matrix or communication is listed for a given time step, the processor is idle. Column headings give the processor tag with its binary representation and the time required for each step is given on the right. A communication step takes approximately $\alpha + \frac{1}{2}p^2\beta$ time, where α is communication startup time and β is communication rate. Each sum then takes approximately $\frac{1}{2}p^2\gamma$ time. There are exactly $d = \log_2 r$ communication-summation steps and the final crossproducts matrix A ends up on processor 0. Note that communication is always between neighbors (binary tags differ by one bit only). Fig. 5 illustrates this communication pattern.

In some applications it may be desirable to have the crossproducts matrix on every node for further computation. These may be an orthogonal decomposition discussed in [12], or any application requiring regression computations with many response variables such as bootstrapping, where each node processor would perform regressions based on the same crossproducts matrix. Putting the crossproducts matrix on each node can be done in exactly the same amount of time as the single crossproducts matrix on node 0 by simply keeping all processors busy during the summing process. This is illustrated in Figures 6 and 7. Note that all communication is still only between neighbors. This communication pattern is known as the *exchange algorithm*, since we exchange data once over each dimension of the hypercube. The exchange algorithm arises in many applications. For example, the Fast Fourier Transform algorithm also requires this communication pattern, but in reverse order.

The total time for this crossproduct computation is

$$\tfrac{1}{2}\frac{np^2}{r}\gamma + (\alpha + \tfrac{1}{2}p^2\beta + \tfrac{1}{2}p^2\gamma)\log_2 r,$$

where $r = 2^d$ is the number of processors. The first part of the expression (local crossproduct computation) decreases linearly with r and the second part (summation) increases logarithmically with r. Thus for a given matrix, there is a maximum number of processors that can be used before the total time begins to increase due to the summation process.

One measure of parallel algorithm performance is *speedup*. This is given by the execution time on a single processor divided by the execution time on r processors.

in or any application requiring regression

Speedup for the above computation is given by

$$\frac{r}{1+\frac{r\log_2 r}{n}\delta},$$

where $\delta = \frac{2\sigma}{p^2\gamma} + \frac{\beta}{\gamma} + 1$. As the number of rows, n, increases, the speedup approaches r. The number of columns, p, affects only δ, which becomes roughly constant for large values of p. Thus speedup of the crossproducts algorithm is affected mostly by n and very little by p.

This algorithm runs an identical program on each node. We start with one partition X_i of X on each node. Providing X_i to each node will differ between different machines. For example on the Intel machine, the host communicates each partition directly to each node via a direct communication link. Provided that the X_i's have been distributed to appropriate nodes, the following node program will compute A on each node.

1. $A = X_i'X_i$
2. *for* $k = 1$ *to* d *do:*
3. send A over dimension k
4. receive B over dimension k
5. $A = A + B$

This is a sequential program that executes asynchronously on each node. Synchronization is achieved by the *send* and *receive*. Both have to be completed before the addition in step 5 takes place, because we need B for the addition and we need to send A before B is added to it.

A FORTRAN implementation of this program was programmed on the Intel iPSC hypercube. A host program (not shown here) sends one partition X_i of X to each node of the hypercube. The node program that runs on each processor is given below.

```
      integer ct, p, d, copen, npar(3)
      real S(60000)
c
c     ------------------------------------
c     open communication channel
c     ------------------------------------
      ct = copen(0)
c
c     ------------------------------------
c     wait to receive local parameters and matrix from host
c     ------------------------------------
      call recvw ( ct , 0 , npar , 12 , ln , nhost , idp )
      n = npar(1)
      p = npar(2)
      d = npar(3)
      lenmat = p*(p+1)/2
      mx = lenmat + 1
      call lrecvw ( ct , 0 , S(mx) , p*n*4 , ln , nhost , idp )
c
c     ------------------------------------
c     compute local crossproduct
c     ------------------------------------
      call crosp ( S(1) , S(mx) , n , p )
c
c     ------------------------------------
c     add crossproducts over each dimension
c     ------------------------------------
      do 10 k = 1,d
        kneibor = ieor ( mynode() , 2**(k-1) )
        call lsendw ( ct , k , S(1) , lenmat*4 , kneibor , 0 )
        call lrecvw ( ct , k , S(mx) , lenmat*4 , ln , node , idp )
        call add ( S(1) , S(mx) , lenmat )
   10 continue
c
      stop
      end
```

Both A and X_i are stored in the array S starting in locations 1 and mx, respectively. The storage locations for X_i are used for B once X_i is no longer needed. Both A and B are symmetric, so only their upper triangle is stored. The subroutines *crosp* and *add* are the usual sequential routines that compute the crossproduct of a matrix and add two matrices, respectively. The *recvw* is a FORTRAN callable communication routine of the iPSC. The *lrecvw* and *lsendw* are FORTRAN subroutines distributed by Intel for communicating long messages on the iPSC. The second parameter in these routines is the message type, which is used to select incoming messages. That is, any incoming message is stored by the node operating system, but a message is read by the program only when that type of message is requested. The function *mynode()* returns the tag of the node where the program is executing. The exclusive-or function *ieor*, applied to *mynode()* and $2^{**}(k-1)$, flips the $k-th$ bit of *mynode()* giving the tag of dimension k neighbor. Synchronization is achieved by the communication routines, which halt the program until a message is sent or until a message of appropriate type is received.

Timing the crossproducts program for various matrix sizes and numbers of processors on the Intel iPSC hypercube produced Table 2. The resulting speedup graphs are given in Fig. 8. As expected, Fig. 8 shows that speedup depends mostly on n and very little on p. Both Table 2 and Fig. 8 show that the use of more than approximately $n/10$ processors produces little additional reduction in execution time, thus efficient use of a hypercube requires that the application is "large enough". This behavior is typical of many parallel algorithms.

Table 2
execution time in seconds for r processors

	matrix size								
	n	120	200	500	1000	100	200	400	100	200
r	p	100	100	100	100	50	50	50	200	200
1		46.7	77.7	193.8	388.0	9.9	19.6	39.2	155.1	310.0
2		23.9	39.4	97.5	194.4	5.1	9.9	19.7	79.5	156.6
4		12.6	20.3	49.4	97.8	2.7	5.1	10.0	42.3	80.9
8		7.1	11.0	25.7	49.8	1.6	2.8	5.2	25.2	43.7
16		4.7	6.7	14.1	26.1	1.1	1.8	2.9	17.3	26.6
32		3.7	4.9	8.3	14.6	.9	1.3	1.8	14.0	18.7
64		3.2	4.0	5.5	8.6	.9	1.0	1.3	12.3	15.4

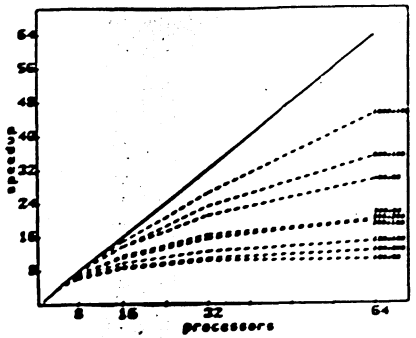

Fig. 8. Speedup curves for various matrix sizes. Solid line shows perfect speedup.

Many computationally intensive methods in statistics can potentially benefit from parallel computation. The crossproduct computation was selected only to illustrate how to program a hypercube. The flexibility of a hypercube makes it ideal for testing parallel algorithms. Perhaps the greatest advantage of the hypercube parallel architecture is that it has been commercially available for almost two years and that a large amount of work has already been spent in developing algorithms and subroutine libraries such as those discussed in [4], [6] and [9].

Acknowledgments. I am grateful to several members of the Computer Science group in our Mathematical Sciences Section of the Oak Ridge National Laboratory for numerous discussions and helpful comments on this paper. This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400.

REFERENCES

[1] T. H. Dunigan, *A Message Passing Multiprocessor Simulator*, Tech Rept. ORNL/TM-9966, Oak Ridge National Laboratory, Oak Ridge, TN 37831 (1986).

[2] T. H. Dunigan, *Hypercube Performance*, Hypercube Multiprocessors 1987, ed. M. T. Heath, SIAM, Philadelphia, 1987, (to appear).

[3] M.J. Flynn, *Very high speed computing systems*, Proceedings of the IEEE 14 (1966), pp. 1901-1909.

[4] G. C. Fox and W. Furmanski, *Optimal communication algorithms on the hypercube*, unpublished Caltech report C^{3}P-314, California Institute of Technology, Pasadena, Calif., July 1986.

[5] G. C. Fox and S. W. Otto, *Algorithms for concurrent processors*, Physics Today, May 1984, pp.50-59.

[6] G. C. Fox, S. W. Otto and A. J. G. Hey, *Matrix algorithms on a hypercube I: matrix multiplication*, Parallel Computing 4 (1987), pp. 17-31.

[7] Karen A. Frenkel, *Evaluating two massively parallel machines*, Comm. ACM 29 (1986), pp. 752-758.

[8] John P. Hayes, Trevor Mudge, Quentin F. Stout, Stephen Colley, and John Palmer, *A microprocessor-based hypercube supercomputer*, IEEE Micro, October 1986, pp. 6-17.

[9] Oliver McBryan and Eric F. van de Velde, *Hypercube algorithms and implementations*, SIAM J. Sci. Stat. Comput. 8 (1987), pp. s227-s287.

[10] C. L. Seitz, *The Cosmic Cube*, Comm. ACM 28 (1985), pp. 22-23.

[11] J. S. Squire and S. M. Palais, *Programming and design considerations for a highly parallel computer*, Proceedings of Spring Joint Computer Conference 1963, pp. 395-400.

[12] G. W. Stewart, *Communication in parallel algorithms: an example*, Computer Science and Statistics: Proceedings of the 18th Symposium on the Interface, ed. Thomas J. Boardman, ASA, Washington, 1986, pp. 11-14.