Abstract. In this paper we consider several algorithms for exchanging data among processors in a hypercube network. The data transfer problems considered are those arising from classical numerical algorithms such as Gaussian elimination, conjugate gradient methods, and the N-body problem. We propose some estimates for the timings of the various algorithms which reveal that multiprocessors based on the hypercube topology can be very efficient in performing data exchange operations.

Data Communication in Hypercubes

Youcef Saad and Martin H. Schultz

Research Report YALEU/DCS/RR-428
October 1985

This work was supported in part by by ONR grant N00014-82-K-0184 and in part by a joint study with IBM/Kingston

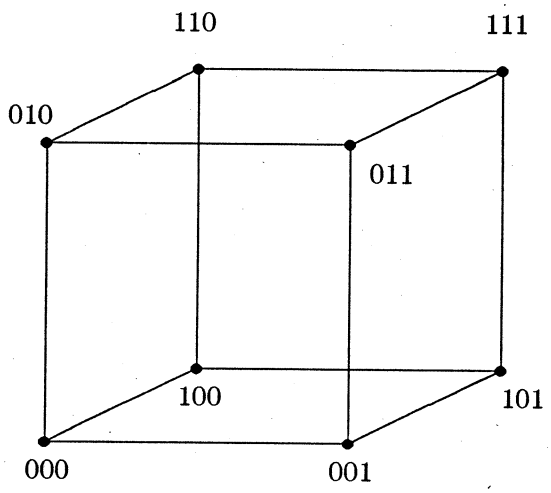

Figure 1: 3-D view of the 3-cube.

1. Introduction

An important factor in the efficiency of loosely coupled parallel computers, is the effectiveness with which data can be exchanged among its many nodes. The speed of communication among the different components of a message-passing machine may well determine the actual effectiveness of a given algorithm. In this paper we will concentrate on the hypercube topology which has recently become a widely accepted interconnection network, and will examine several algorithms for data transfers among processors.

An n-cube network or n-dimensional hypercube, consists of 2^n identical nodes that are interconnected to each other in such a way that each node has n neighbors. The rigorous description of the interconnection involves a binary numbering of the nodes: two nodes are connected if and only if their binary numbers differ in one and only one bit. Thus, for $n = 3$, an n-cube can be represented as an ordinary cube in three dimensions where the vertices are the $8 = 2^3$ nodes of the 3-cube, see Figure 1.

The hypercube network has been attracting much attention because of its powerful topological properties [5] and its recent development into a commercial product by Intel (iPSC series). In particular, many of the classical interconnects such as two-dimensional or three-dimensional meshes (in fact arbitrary dimension meshes) can be imbedded in it. Another appealing feature of the hypercube is its homogeneity and symmetry properties. In contrast with the tree structure or the shuffle exchange structure, no node plays a special role. This facilitates algorithm design as well as programming.

Seitz [8] described a hypercube machine, the Cosmic Cube, which is utilized at CalTech and for the first time presented some details of software and applications for a machine based on the

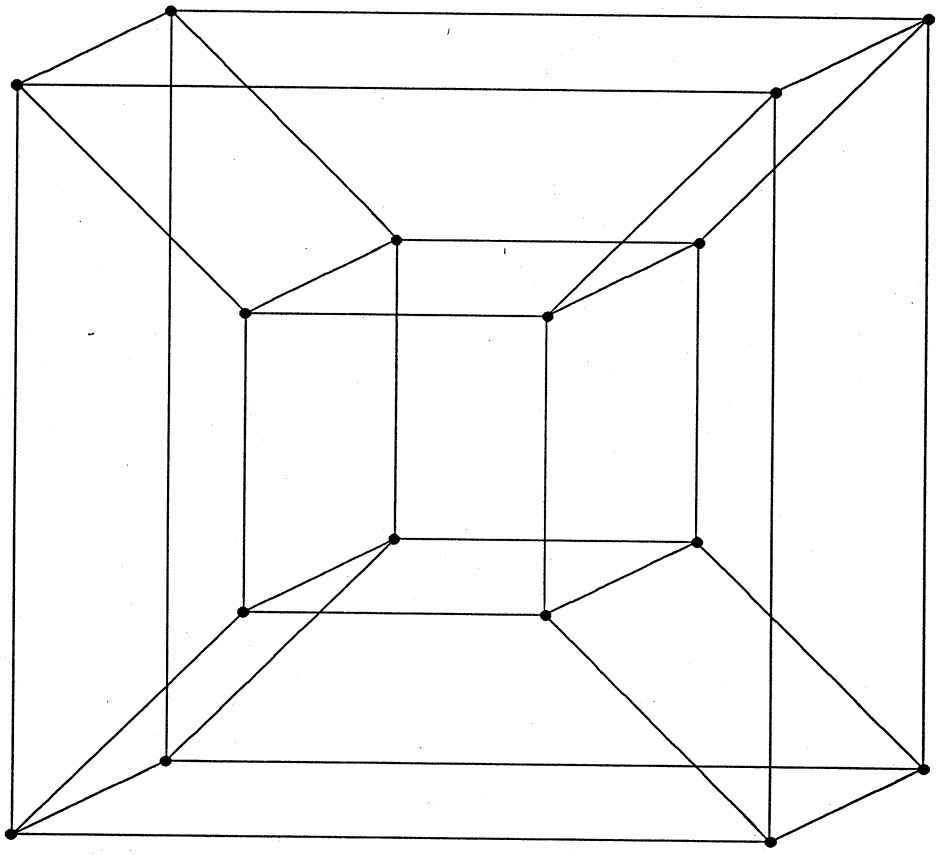

Figure 2: 3-D view of the 4-cube.

hypercube architecture. In [5], we analyzed the topology of the hypercube regarded as a graph and showed many of its intrinsic properties. In [6] we presented and compared an elimination algorithm for solving banded linear systems, to similar methods for a ring of processors and a grid of processors. As expected the comparison showed the superiority of the hypercube interconnect over the simpler interconnects, due to the faster communication. Bhuyan and Agrawal proposed a generalization of the hypercube topolgy[1] and proved a few optimality properties.

In this paper we present several algorithms for transferring data among the nodes of a loosely coupled machine based on a hypercube interconnect. Data transfers are performed by passing messages between nearest neighbors. Thus, a data packet which must travel from node A to node B must cross a sequence of nodes starting at node A and ending at node B. The length of the shortest path between any two nodes A and B in an $n-$cube is at most n.

We examine the important types of data transfer operations one by one. In order to study the complexity of the different algorithms and their efficiency we use the following model, see [2, 6]:

1. Moving a vector of length m from one node to a neighbor takes the time

$$\beta + m\tau. \tag{1.1}$$

   The constant β will referred to as the communication start-up or latency and the constant τ will be referred to as the elemental transfer time. Note that it is usually the case that $\beta >> \tau$.

2. It takes the same time to move the same data item from one node to *any number of its n* neighbors.

An important consideration is whether it is possible in practice to simultaneously use the n channels of a given node to send different data items in each of them. This depends solely on the design of each node. It is clear that without such a capability the high total bandwidth of the machine will not be fully exploited. We should point out that as n increases, each node becomes more difficult to design and fabricate due to larger number of connections. In fact this consitutes the most serious limiting factor with higher-dimensional hypercubes and is often cited as their main drawback [4]. For this reason it is only natural to expect that some advantage must be gained in return for building the more complex higher-dimensional hypercubes. As will be seen, the capability of sending data simultaneously to n neighbors allows many algorithms to take full advantage of the rich interconnection features of the hypercube. For example when one node sends data to another node, then if this assumption holds it will be shown that the total transfer time can be divided by a factor of n by simply splitting up the data into n equal size packets and routing it through *n parallel* paths. Therefore, throughout the paper we will assume that it is possible to overlap n different data transfers from any given node to its n neighbors. We will refer to this assumption as the *communication overlapping assumption.* However, we will always indicate whether this assumption is important for a given algorithm.

Throughout the paper, powers applied to the binary bits 0 or 1 refer to concatenation. The particular node whose label is 0^n is often denoted by O.

2. Moving data from one node to all other nodes

Transferring data from one processor to all other processors, or to a subgroup of the processors is an important operation in many well-known algorithms. An outstanding example is the Gaussian elimination algorithms [2, 6] in which one must send the pivot row (or parts of it) to several processors in order to perform the row eliminations. We will often use the term broadcast by abuse of language to refer to such data communication operation. Sometimes it is only one data element that must be broadcast, such as in the conjugate gradient algorithm in which some scalars resulting for inner-products of two vectors must be made available to all processors in order to update the current approximation.

The following is a summary of what has been fully developed in [6]. A simple algorithm for sending *one* element from the processor labelled O, to all others in an n-cube is based on the recursive construction of n-cubes from lower-dimensional cubes as was seen in [5]: at the j^{th} step, $j = 2, \ldots n$, all the nodes whose binary label is of the form $0^{n-j+1}a$ where a is any $(j-1)$-bit binary number send the data item (received in previous step) to their neighbors $0^{n-j}1a$.

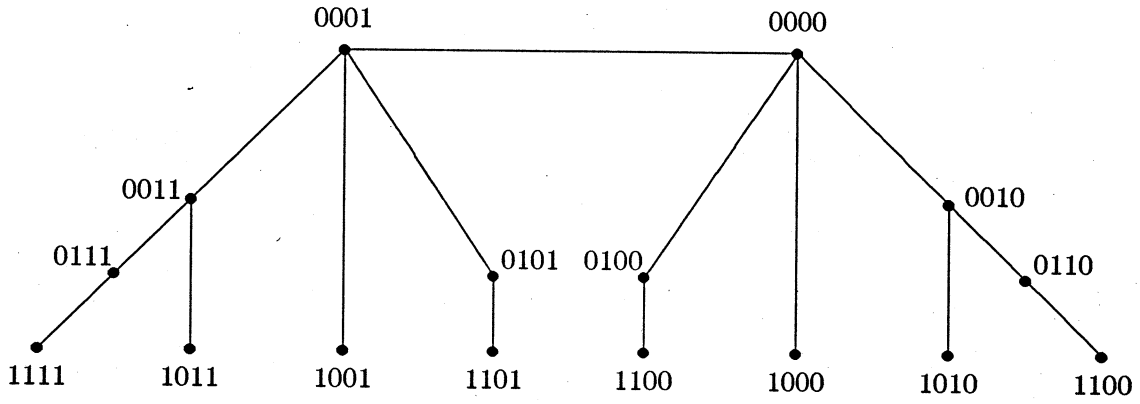

Figure 3: Spanning tree of the 4-cube.

ALGORITHM: Hypercube Broadcast

(1) *Start:* Send data item from processor 0^n to processor $0^{n-1}1$.

(2) *Loop:* For $j = 2, 3, \ldots n$ do

All processors numbered $0^{n-j+1}a_j$, where a_j is any $(j-1)$-bit binary number move data item in parallel to processor $0^{n-j}1a_j$.

It is instructive and helpful for the algorithms to be derived later to illustrate the data paths of the above algorithm with the spanning tree of the hypercube, shown in Figure 3, see [5]. The first step of the hypercube broadcast takes the data from node 0^n to node $0^{n-1}1$. Afterwards, each level of the tree in turn reads the data from the upper levels. The above algorithm consists of a total of n steps all requiring the same amount of time. Therefore we have:

Proposition 2.1. *A data item can be transferred from one node to all other nodes of a hypercube in a total of n steps and in an approximate time of*

$$n(\beta + \tau) \tag{2.1}$$

When sending a vector of length m from one node to all others, we can pipeline the data transfer as suggested in [2, 6]. Pipelining consists of sending data packets of identical size, one after the other in a assembly line manner: each node of the path simultaneously receives the next

packet from the previous node and sends its previously received packet to the next node of the path. There is an optimal packet size which depends only of the parameters m, τ, β and the length of the path. In this case the length of the longest path is n, and we showed the following result in [6].

Proposition 2.2. *The optimal time in which a vector of length m can be sent from one processor to all other processors in an n-cube using pipelining is given by:*

$$t_{opt}(m,n) \approx \left(\sqrt{m\tau} + \sqrt{(n-1)\beta}\right)^2. \tag{2.2}$$

In the above discussion we have particularized the node $O = 0^n$, by describing how to perform a broadcast operation from it. If we want to broadcast from another node, say node A we can simply renumber the nodes so that node A becomes node O and the new numbering is consistent with the hypercube numbering, i.e., each neighbors still have their new labels differ by one bit. Clearly, this is achieved by an exclusive or of node label with label A. This argument will often be utilized in the rest of the paper.

3. Moving data between two nodes

Let A and B be any two nodes of the n-cube and consider the problem of sending data from node A to node B. Clearly, a path of length at most n between the two nodes is obtained by crossing successively the nodes obtained by modifying the bits of A one by one in order to transform the label A into the label B [5]. Recalling that the number of different bits between two binary numbers A and B is called the Hamming distance between those two numbers, the minimal path length between A and B is precisely the Hamming distance $H(A,B)$ between the binary labels A and B.

Thus one particular path from $A \equiv a_1a_2..a_n$ to $B \equiv b_1b_2..b_n$, where a_i, b_i are zeroes or ones, corresponds to correcting the first differing bit, then the second and so on to the last differing bit. Assuming, without loss of generality, that A and B differ in their i leading bits then this path is:

$$\begin{aligned} A = &\text{node } 0 = a_1a_2a_3\ldots a_ia_{i+1}\ldots a_n; \\ &\text{node } 1 = b_1a_2a_3\ldots a_ia_{i+1}\ldots a_n; \\ &\text{node } 2 = b_1b_2a_3\ldots a_ia_{i+1}\ldots a_n; \\ &\ldots\ \ldots \\ B = &\text{node i} = b_1b_2..b_ia_{i+1}\ldots a_n. \end{aligned}$$

The generalization to the case where the different bits are not necessarily the leading ones is straightforward.

It is important to know whether there are different paths between A and B that can be simultaneously taken in order to speed transfers of large amounts of data between these two nodes. In order for this to be possible the paths must not cross each other, i.e. they must not have commun nodes, except for nodes A and B. We will refer to such paths as parallel paths. So the above question can be reformulated as follows: *how many parallel paths are there between any two nodes A and B?*

A simple look at the above path between A and B reveals that there is no reason to start by correcting the first differing bit. More generally we might start correcting the j^{th} bit (where $j \leq i$) then the $(j+1)^{st}$ bit and so forth until the i^{th} bit is reached, after which we correct in turn bits $1, 2, \ldots (j-1)$. We can thus define i different paths. It was shown in [5] that these paths are parallel.

In particular, when $i = n$ this result is optimal since there cannot exist more than n parallel paths between two nodes. When $i < n$ we have shown in [5] that there are $n - i$ additional paths that are parallel to each other and parallel to the previous i paths. The length of each of these additional paths is is $i + 2$. The following proposition summarizes these results.

Proposition 3.1. *Let A and B be any two nodes of an n-cube and let $i = H(A,B)$ their Hamming distance. Then there are n parallel paths between the nodes A and B. Moreover, i of these paths are of length i and the remaining ones are of length $i+2$.*

The above result is important as it will allow us to take advantage of the communication overlapping assumption, see Section 1.

We now wish to estimate the time it takes to transfer data between any two nodes. If only one path is used, one can split the data into ν equal packets and pipeline the data transfer along the path [7, 2]. There is an optimal number of packets given by

$$\nu_{opt} \equiv \sqrt{i\frac{\tau}{\beta}},$$

where i represents the length of the path, and the corresponding optimal time is given by

$$\begin{aligned} t_{opt}(m,i) &\approx m\tau + (i-1)\beta + 2\sqrt{(i-1)m\tau\beta} \\ &= \left(\sqrt{m\tau} + \sqrt{(i-1)\beta}\right)^2. \end{aligned} \tag{3.1}$$

Assume now that the design of each node is such that it is capable of sending one different message to each of its n adjacent nodes at once, i.e., that it is capable of overlapping communication is its n channels. This capability will make it possible to use the n parallel paths of the above proposition simultaneously. The data is first split in n equal parts. Then each part is moved along one of the n parallel paths. Pipelining can be used along each path. It is clear that since the longest path has length $H(A,B)+2$, if $H(A,B)<n$ and length $H(A,B)$ otherwise, we have proved the following result.

Proposition 3.2. *It is possible to send m data elements from node A to node B in time*

$$t_{Par,m,i} = \left(\sqrt{\frac{m}{n}}\tau + \sqrt{(i-1)\beta}\right)^2.$$

where $i \equiv H(A,B)+2$ if $H(A,B)<n$ and $i = H(A,B)$ otherwise.

Observe that the increasing bandwidth of the system which is associated with increasing number of edges per node, is reflected by a division of the total data length m by n.

4. Moving data from every node to every other node

Throughout this section we consider the data permutation problem in which

every processor sends a block of data of the same size m to every other processor.

This can be considered as a multi-broadcast operation, since every node needs to broadcast its data to all the others. An important situation where this type of data exchange arises is when a matrix is stored by rows in the k processors and one wants to compute repeatedly a matrix by vector multiplication, such as happens in the power iteration $v_{j+1} = Av_j$. Assuming that the matrix A is stored by blocks of N/k rows each (N may be assumed to be divisible by k), and that the vector v_j is initially in every processor, each step of the above iteration can be split in two separate stages, an arithmetic stage and a data transfer stage. In the arithmetic stage every node computes the product of its part of the matrix A by the vector v_j which is initially available in each processor. At the end of this stage, every processor holds a subvector of length N/k of the result $v_{j+1} = Av_j$. The purpose of the second stage is to make the whole vector v_{j+1} available in every processor in

order to enable the computation of the next iterate $v_{j+2} = Av_{j+1}$. Thus every subvector of size N/k of v_{j+1} which is available in some processor must be broadcast to every processor.

Another example of this sort of "total data exchange" arises in numerical methods for solving the N-body problem in mechanics where each of N bodies exchanges information with all others in order for one integration step to proceed. If we assume that there are $N = 2^n$ bodies, one per node of the hypercube, then each integration step requires an extensive data permutation so that $2^n - 1$ information packets are to be received by every node, one from each other node.

A few algorithms are presented below with improving quality.

4.1. Sequential Broadcasting and Daisy-Chaining

As a first algorithm we consider the naive approach of broadcasting the data from each node in turn using the Hypercube Broadcast Algorithm of section (2.1). We will refer to this method as the Sequential Broadcasting Algorithm (SBA), for reasons that will be clarified later. Since we perform a Hypercube Broadcast from each of the 2^n nodes in turn, this will require a total time of

$$t_{SBA} = 2^n \left(\sqrt{m\tau} + \sqrt{(n-1)\beta}\right)^2. \qquad (4.1).$$

Since many links are idle most of the time, it is clear that we might expect to find a better algorithm by trying to utilize the data communications links more efficiently.

A second method is based on mapping the ring into the hypercube, and exchanging the data along the ring in a daisy-chain manner. As was seen in [5], a ring can be mapped into the hypercube with the help of Gray codes. We choose a direction of data movement in the ring. At every step, each processor sends to its next processor (in the ring) the data packet that it has just received and simultaneously receives the next packet from the previous processor. After $2^n - 1$ steps, every processor will have received all other data packets. Hence, the total time required for performing this Daisy-Chaining Algorithm is

$$t_{DC} = (2^n - 1)(\beta + m\tau). \qquad (4.2)$$

4.2. Alternate Direction Exchange Algorithm

The Alternate Direction Exchange Algorithm consists of tearing the n-cube into two $(n-1)$-cubes in each of the n directions in turn and exchanging the data between the nodes of the two lower dimensional cubes. We recall that two nodes are said to be opposite to each other along the i^{th} direction if their binary numbers differ only in the i^{th} bit.

ALGORITHM ADEA: Alternate Direction Exchange Algorithm

(1) *Start:* Each node with leading bit one exhanges its (one) data packet with its opposite node in the 1^{st} direction.

(2) *Loop:* For $i = 2, 3, \ldots n$ do

Each node whose i^{th} bit is one exhanges all its accumulated 2^{i-1} data packets with its opposite node in the i^{th} direction.

We notice that each data packet seen separately will successively cross the same links as those of the Sequential Hypercube Broadcast Algorithm. In other words this method can be regarded

as a parallel version of SBA. In particular, this proves that the algorithm realizes the desired total permutation of data.

We now consider the cost of this method. At the i^{th} step of the algorithm we twice move blocks of $2^{i-1}m$ elements in parallel along the edges in the i^{th} direction, which takes the time $2(\beta+2^{i-1}m\tau)$. Summing this over the n steps we get the following total time for executing Algorithm ADEA

$$t_{ADEA} = 2n\beta + (2^{n+1} - 2)m\tau \qquad (4.3).$$

Observe that at any step, only the 2^{n-1} edges linking opposite nodes along one direction are active. A clear consequence is that there might still be room for an n-fold improvement, provided each node is able to communicate in all directions at once, which is the motivation of the algorithms presented in the next section.

4.3. The Total Exchange Algorithm

Our next algorithm, called Total Exchange Algorithm (TEA), forces every edge to be active carrying data all the time. This algorithm uses the assumption of overlapping in communication in an attempt to derive a faster method than the ones described above. A schematic description of the algorithm follows:

For $j = 1, 2 \ldots n$ do:

Every processor reads from its n neighbors *one copy* of each data packet that it does not yet hold.

It is easier to think of the algorithm in terms of the activity of the links rather than of the nodes. With this viewpoint each step consists in exchanging data along *all links in parallel* between the nodes that these links connect. As is to be expected, the organization of the method is somewhat complicated. For this reason, we will first present an algorithm which is simple to describe but which is non-optimal as it exchanges more data than is necessary. We will then "optimize" this algorithm by having the links carry only the minimal information needed by the other nodes.

We will refer to the data packets and the node ids where they come from during the data exchange as *tokens*. Each token can be regarded as a box carrying the data packet itself plus a label T consisting of the binary number of the node where it comes from. The algorithms to be proposed will exchange these tokens along certain edges, according to their labels.

Consider first the result of one step of the total exchange algorithm, in which every edge is used to exchange data between its adjacent nodes: every node will thus receive the (one) token of each of its neighbors, i.e., of all the nodes that are at distance one from it. We can successively repeat the operation by having every node get from each of its neighbors the tokens that it does not already have. For example, in the second step each node A will get from its neighbors all the tokens that they have just received except obviously the token A. Clearly at step i, every node will have the data of the nodes that are at distance $\leq i$ from it. In order to systemize the data exchange between two neighboring nodes, say A to B, it suffices to proceed as follows:

noindent At step i, scan all tokens T contained in node A and send token T to node B if and only if $H(T, B) = i$, where $H(A, C)$ is the (Hamming) distance between A and C.

With this we can formulate the algorithm TEA1.

ALGORITHM TEA1: Total Exchange Algorithm (Non-Optimal)

For $i = 1, 2, \ldots n$ Do:
Along every edge (A, B) Do in parallel:

(1) *Send from node A to node B all tokens T such that* $H(T, B) = i$.
(2) *Send from node B to node A all tokens T such that* $H(T, A) = i$.

As can be seen from the simple example of the 3-cube much data is unnecessarily replicated in this exchange, i.e., every node will get several copies of the same token. This means that this algorithm is different from the (optimal) total exchange algorithm sketched above. For example, node number 000 will successively get the tokens

$$001, 010, 100,$$

in the first step and

$$\{101; 011\}, \quad \text{From node } 001,$$
$$\{110; 011\}, \quad \text{From node } 010,$$
$$\{110; 101\}, \quad \text{From node } 100,$$

in the second step. Thus, in the second step each token whose label contains two ones is read twice into node 000. The reason for this is simply that any token that is originally at distance 2 from 000 will follow the two different paths described in Section 3. Therefore, after step i of this algorithm each token that is at distance i from some node will be read i times in this node. This is verified in the above example.

Next we wish to estimate the time required to execute Algorithm TEA1. At step i of algorithm TEA1, the tokens T that must be moved from node B to node A are those that have been received in B in step $i - 1$, and whose labels differ from the label A in exactly i bits. As was already observed, this is related to the i paths from node T to node A, when $H(A, T) = i$. The passage of token T from B to A corresponds to the last crossed edge of the path. If A and B differ in their bit number j, this crossing will in fact simply correspond to correcting the j^{th} bit according to the terminology introduced in Section 3. Hence, the j^{th} bit of the token is the same as that of B. Moreover, the label of T differs from the label of B in $i - 1$ different bits. There are exactly

$$\binom{n-1}{i-1}$$

such tokens and therefore the total time required by Algorithm TEA1 is

$$t_{TEA1} = 2\sum_{i=1}^{n}\left[\beta + \binom{n-1}{i-1} m\tau\right],$$

or

$$t_{TEA1} = 2n\beta + 2^n m\tau. \tag{4.4}$$

This is smaller than the time for Algorithm ADEA. In fact, the start-up times are identical for the two methods but the transfer time for Algorithm TEA1 is about half that of ADEA.

We now derive a modification of TEA1 in which we attempt to optimize the data exchange. For this purpose we need to examine more closely the situation at a general step i. Consider the particular node 0^n after $i-1$ steps of the total exchange algorithm. The tokens T whose labels are at distance i from 0^n must be located in the n neighbors of node 0^n and each of them will be represented i times in these nodes, once for each of the i different paths of length i from the original node of the token to the node 0^n. Moreover, we know that the nodes where token T will be at that step are all those whose labels are obtained from the label of token T by replacing all the ones by zeroes except one of them. The nodes thus obtained are the last nodes of the path from node T to node 0^n, the final destination 0^n of the path. In other words, the node having zeroes everywhere except at the j^{th} position will contain all tokens having i ones, one of which is in position j. There are exactly $\binom{n-1}{i-1}$ such tokens.

The problem now is to select from each of these nodes an equal number of tokens to move to node 0^n. Ideally, since each node is represented i times we would like to move only $1/i$ part of all the tokens of each neighbor of 0^n, among all those tokens that are at distance i from node 0^n. For illustration when $n=5$ and $i=3$ the situation after step $i-1$ is as follows:

$$\{10101; 11001; 10011; 01101; 01011; 00111\} \in \text{Node } 00001$$
$$\{10110; 10011; 11010; 01110; 01011; 00111\} \in \text{Node } 00010$$
$$\{10110; 10101; 11100; 01110; 01101; 00111\} \in \text{Node } 00100$$
$$\{11010; 11001; 11100; 01110; 01101; 01011\} \in \text{Node } 01000$$
$$\{11010; 11001; 11100; 10110; 10101; 10011\} \in \text{Node } 10000.$$

In the above example, there are $\binom{5}{3} = 10$ nodes that are at distance 3 from the node 00000. These are scattered, each of them being repeated three times, in the 5 neighbors of node 00000 after step 2 of the total exchange algorithm in the way shown above. Here it is clear that the desired optimality is achieved in step 3 by moving two of the 6 tokens from each of the 5 nodes, so that each token is moved only once from some node to node 00000. For example, one might select the tokens to move to node 00000 as follows.

$$\text{From Node } 00001 : \{10101; 11001\}$$
$$\text{From Node } 00010 : \{01011; 10011\}$$
$$\text{From Node } 00100 : \{10110; 00111\}$$
$$\text{From Node } 01000 : \{01101; 01110\}$$
$$\text{From Node } 10000 : \{11010; 11100\}.$$

We observe that the last four pairs are obtained from the first one by $4 = n-1$ successive (left) shifts. More generally, we can partition the set $A(n,i)$ of all n-binary numbers having i ones and $n-i$ zeroes, into disjoint equivalence classes with the help of the equivalence relation

$$a \ \mathbf{S} \ b \text{ if and only if } a = S^j b \text{ for some } j, \quad 0 \le j \le n,$$

where S is the left cyclic shift operator. In the above example there are two equivalence classes: the classes of 10101 and 11001. Observe that we have selected the representative elements of the classes so that they have a one as their last bit, because we wish to put those elements in node number $e_1 \equiv 0^{n-1}1$. As we shift these representative elements j bits to the left the bit one will

move to the the j^{th} position, i.e., the corresponding token is located in the node number e_j which has zeroes everywhere except in position j.

Denoting by S the cyclic left shift operator, and given a set of representative elements $R(n,i)$ having a one as their last bit, optimality in the total algorithm is achieved by moving at step i of the algorithm, the set $R(n,i)$ from node $e_1 \equiv 0^{n-1}1$ to node 0^n and more generally the set

$$S^{j-1}R(n,i) \text{ from node } e_j \equiv S^{j-1}\, e_1 \text{ to node } 0^n, \quad j = 1,\ldots,n.$$

So far we have particularized the node 0^n and expressed the data movements to that node from its neighbors only. More generally, in order to know which tokens to move from a given node A to its neighbor B, it suffices to return to the particular situation $B = O$ by performing an exclusive OR of the binary label of B with all the binary labels of the tokens contained in A. Thus, with the notation introduced above, one can formulate the general algorithm as follows:

ALGORITHM TEA2: Total Exchange Algorithm (Optimal)

For $i = 1,2,\ldots n$ do:
Along every edge (A,B), where A,B differ in their j^{th} coordinate, do in parallel:

(1) *Send from node A to node B all tokens T such that* $XOR\ (T,B) \in S^{j-1}R(n,i)$

(2) *Send from node B to node A all tokens T such that* $XOR\ (T,A) \in S^{j-1}R(n,i)$

It is important to look at the algorithm from the implementation side. First we point out that both algorithms TEA1 and TEA2 have been expressed in terms of activities of the interconnection links. This needs some clarification. If each of the links of the cube is a duplex line communication in both directions of every link can take place simultaneously and the innermost loop of the above algorithm can be performed in a straightforward way. If not the exchange operation along the edges at the innermost loop in both algorithms TEA1 and TEA2 can be performed as follows. If we define the parity of a node to be even if the nmber of ones in its label is even and odd otherwise, then in a first pass all even nodes send the selected tokens to their odd neighbors, and in a second pass all odd nodes send the selected tokens to all their even nodes, see Figure 4. In an actual implementation of this algorithm for a message-passing machine, each node should appear as executing a loop of n steps each consisting of first sending data through each of its n channels, and then receiving data from all its n channels. Although the algorithm is formulated as if there were some form of synchronization, in reality this is not necessary. A node A will not send its selected tokens T to B until they are all available.

However, there is a slight improvement to the above implementation which we will describe for algorithm TEA1, with an obvious extension to TEA2. Let us assume that the algorithm starts by sending the data from even to odd nodes at the first half step and then from odd to even nodes at the second half step. In the very first step the neighbors of the even node O, which are odd, will have received in addition to the data sent from node O all the tokens that are at distance 2 from O. If we applied algorithm TEA1, as it is described above, the next half step would consist only of sending from any odd node A to its even neighbor B all the data that it contains which is at distance one from B. Node O would then get the tokens of its neighbors. However, nothing prevents us from also sending the data that is at distance 2, which is already in the neighboring nodes. Clearly, by doing so we are grouping two half steps of the original algorithm together, and

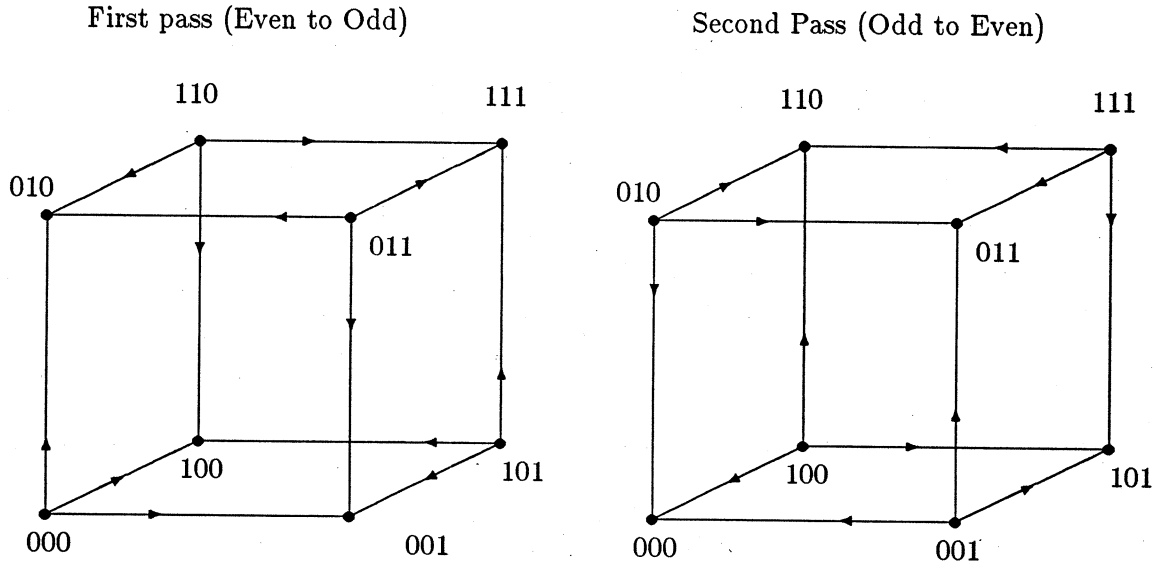

Figure 4: Organization of total data exchange algorithms in a 3-cube.

therefore we will save about half the communication start-ups. Thus the main loop of the above algorithm becomes,

For $i = 0, 1, \ldots, \lceil \frac{n-1}{2} \rceil$ do:

(1) *Send from every even node A to each of its neighbors B that differs from A in the j^{th} bit, all tokens T such that XOR $(T, B) \in S^{j-1}R(n, 2i)$ or XOR $(T, B) \in S^{j-1}R(n, 2i+1)$*

(2) *Send from every odd node B to each of its neighbors A that differs from B in the j^{th} bit, all tokens T such that XOR $(T, A) \in S^{j-1}R(n, 2i+1)$ or XOR $(T, A) \in S^{j-1}R(n, 2i+2)$*

In the algorithm without grouping, each exchange step requires that we send in parallel a total of

$$\left\lceil \frac{1}{i} \binom{n-1}{i-1} \right\rceil$$

tokens in each direction. Thus the total cost of each step is such that

$$t_i \leq 2 \left[\beta + [\frac{1}{i} \binom{n-1}{i-1} + 1] m\tau \right]$$

Observing that $\frac{1}{i}\binom{n-1}{i-1} = \frac{1}{n}\binom{n}{i}$, and summing up over $i = 1, 2, \ldots n$ we get the following upper bound for the total execution time of Algorithm TEA2:

$$t_{TEA2} \leq 2n\beta + \left[\frac{2^{n+1}}{n} + 2n \right] m\tau. \tag{4.5}$$

When grouping two consecutive half-steps, the total number of steps is reduced to $2\lceil\frac{n}{2}\rceil$ and therefore the term $2n\beta$ which reflects the contributions from start-up times changes to at most $(n+1)\beta$. Note that the contribution of communication start-up time can be quite important because as we mentioned before, it is usually the case that the communication start-up β is very large as compared with the elemental transfer time τ. Thus a reduction by a factor of nearly two of this contribution is nonnegligible especially in case the amount of data m is small.

We emphasize that there is an important overhead in the algorithm, not accounted for in the above formula, since token labels must be examined before the tokens are transferred. This means that the algorithm may be ineffective for small amounts of data.

5. Scattering and Gathering of Data in a Hypercube.

Many algorithms require moving k pieces of data of the same length from all nodes to one particular node. We refer to this data transfer operation as "data gathering." One important example is that of the computation of inner-products. If two vectors of length N are equally distributed in the k nodes and their partial inner-products are computed in each node, one must sum these partial inner-products to obtain the result. One way of accomplishing this is to send the k partial inner products in one processor and add them in that processor.

The dual operation of data gathering is "data scattering" in which a particular processor sends pieces of data of equal size to all processors. This is similar to broadcasting but instead of sending the same data packet to every node, we now want to send $k-1$ different packets of equal size m, one to each different node. Algorithms similar to broadcasting can therefore be easily derived. An important and useful observation is that gathering and scattering are no different in concept: any algorithm for gathering can be translated into its dual algorithm for scattering and vice-versa by simply *reversing the data paths.*

As it turns out, gathering algorithms are slightly easier to formulate so we will first propose a Gathering Algorithm which is obtained as a generalization of the Hypercube Broadcast algorithm, by looking at the reverses of the data paths of that algorithm. It gathers one data packet of size m from each processor to node 0^n.

ALGORITHM: Hypercube Gather Algorithm

For $j = n, n-1, n-2, \ldots 1$ do:
All processors numbered $0^{n-j}1a_j$, where a_j is any $(j-1)$-bit binary number move in parallel their data packets accumulated from the previous steps to processors $0^{n-j+1}a_j$, respectively.

The Hypercube Gather algorithm consists of a total of n steps. Assuming that the data items all have the same length m, at the j^{th} step the algorithm transfers a vector of length $2^{j-1}m$ between two nodes, since the amount of data collected per node doubles at every step. Hence the total time required for executing a Hypercube Gather is

$$t_{HG} = \sum_{j=1}^{n}(\beta + 2^{j-1}m\tau) = n\beta + (2^n - 1)m\tau.$$

To derive the dual scatter algorithm it suffices, as we previously suggested, to consider the reverse path of each data packet in the Hypercube Gather Algorithm. The only difficulty is notational. In the above algorithm this notational difficulty was avoided by stating that at a given

step some nodes send the data packets which they have accumulated from previous steps of the algorithm, without being specific about which those packets might be. Let us then start with a vector $v = (v_0, v_1, \ldots v_{k-1})$ in processor 0^n. We will assume that each subscript j of v_j is a binary number that indicates the processor number where v_j should be transferred. In particular, the vector v_0 is not to be moved anywhere.

To better understand the scatter algorithm we refer to the spanning tree of Figure 3. The difference between the broadcast algorithm and the scatter algorithm is that the first algorithm sends the same data to all processors while the second sends a different data packet from node $O = 0^n$ to each processor.

The key observation is that the label of each child of a node of level i of the tree has the same i trailing bits as the label of its father. Changing from level i to level $i+1$ corresponds to simply replacing the bit in position $i+1$ which is zero at level i by the bit one. At first we start by sending from node 0^n to node $0^{n-1}1$ all subvectors v_j whose labels end with the bit one. Now both of these nodes have one half of the total data. More generally, at a given step i, the nodes of level i of the tree will all have the subvectors whose labels have i trailing bits which agree with the trailing bits of the label of the partciclar node to which they belong. For a data packet labeled j to reach it destination which is the node with label j, we must send it downward, to the node of the next level which has a label with more bit agreeing bit (namely the bit in position $i+1$), and we repeat this process until its destination is reached.

ALGORITHM: Hypercube Scatter Algorithm

For $j = 1, 2, \ldots n$ do:
All processors numbered $0^{n-j+1}a_j$, where a_j is any $(j-1)$-bit binary number, move all sub-vectors v_x with labels $x = b_{n-j-1}1a_j$ where b_{n-j-1} is any $(n-j-1)$ bit binary number in parallel to nodes $0^{n-j}1a_j$.

At the j^{th} step of the algorithm, every node A with a label of the form $0^{n-j+1}a_j$, i.e., every node of the first j levels, contains the subvectors with labels xa_j, x being an arbitrary $(n-j+1)-$binary number. Each node A keeps the subvector that is destined to it and drops those for which x has a trailing bit one to the lower levels. We observe that as the algorithm progresses, the amount of data to be moved shrinks by a factor of 2 at each step, instead of doubling as in the Hypercube Gather Algorithm. There is no difference in the timings of the two dual methods.

Proposition 5.1. *A scatter or a gather operation of* $(2^n - 1)$ *vectors of length* m *each can be achieved in time*

$$t_{HG,HS} = n\beta + (2^n - 1)m\tau. \tag{5.1}$$

6. Multiscatter-gather or Data Transposition

In the present section we consider the problem of simultaneously gathering in each processor data packets from every other processor. In other words, this problem can be considered as a generalization of the data gathering problem of the previous section. It can also be regarded as a generalization of the data scattering problem by considering the dual algorithms.

Forming the transpose of an N x N matrix A is a very good illustration of this type of data permutation. Assuming that A is stored in rectangular blocks of N/k *rows* in each processor, where

$k \equiv 2^n$ is the total number of processors, we would like to permute the data so as to store the matrix in blocks of N/k *columns* in each processor. Let us partition the matrix in square blocks of size N/k x N/k, and denote by $A_{i,j}$ the entry in position i,j of the block matrix thus defined. All blocks $A_{i,j}$, where i is fixed and j varies from 1 to N/k belong to processor number i. We would like to gather each block $A_{i,j}$, where j is fixed from processors $i, i = 1,\ldots,j-1,j+1,\ldots k$ and transfer it to processor number j. Moreover, we would like to do this for $j = 1,\ldots k$. The term multigather describes the situation well. Clearly, one can see the problem as a multiscatter as well, by considering that the blocks $A_{i,j}, j = 1\ldots k$, must be scattered from processor i to processors $j, j = 1\ldots k,\ j \neq i$, respectively.

We will describe our algorithms in terms of transposing a square matrix of size N. A naive algorithm based on the above observations is simply to perform k scatter or gather operations. Each of the blocks is of size (N/k) x (N/k) and therefore the length of each packet to scatter (or gather) with the algorithms of the previous section is N^2/k^2. The total time for this Sequential Multi-Gather Algorithm is

$$t_{SMG} = k\left[\frac{(k-1)N^2}{k^2}\tau + n\beta\right] \approx N^2\tau + n2^n\beta.$$

The next algorithm is an analogue of the Alternate Direction Exchange algorithm of Section 4. The principle is to exchange data across opposite edges along the n directions in turn. Consider again the binary numbering of the blocks $A_{i,j}$ where for convenience we assume here that the subscripts i and j start at zero, i.e., $0 \leq i,j \leq k-1$. To formulate the algorithm we need to introduce the function $b_i(X)$ whose value is the bit in position i of the $n-$bit binary number X. We denote by $\bar{X}^i$ the binary number obtained from X by complementing the i^{th} bit of X. This algorithm was already presented in [3].

ALGORITHM MTADEA: Matrix Transposition by Alternate Direction Exchange

(1) *Loop:* For $i = 1,2,\ldots,n$ do: Each node X such that $b_i(X) = 1$, exchanges with node $\bar{X}^i$ all blocks $A_{X,Y}$ and $A_{\bar{X}^i,Y^i}$ where Y is any binary label such that $b_i(Y) = 0$.

(2) *Final step:* Transpose the resulting $A_{i,j}$ in each node.

Thus, the first step of the algorithm consists of exchanging the matrices that are in upper right and lower left positions of the large 2 x 2 block matrix obtained from A by splitting it into 4 equal parts. Afterwards, we exchange matrices whose size is still $N/2$ but these matrices are scattered among other processors in a more complicated way. Hence each step costs

$$2\left[\frac{N^2/4}{k/2}\tau + \beta\right] = \frac{N^2}{2^n}\tau + 2\beta$$

and upon summation over the n steps we get the total time:

$$t_{TMADEA} = n\left[\frac{N^2}{2^n}\tau + 2\beta\right].$$

Acknowledgement We are indebted to Stan Eisenstat for making a suggestion that lead to the improved version of the total exchange algorithm involving fewer communication start-ups.

References

[1] L. N. Bhuyan, D.P. Agrawal, *Generalized Hypercube and Hyperbus structures for a computer network,* IEEE Trans. Comp., C-33 (1984), pp. 323–333.

[2] I. Ipsen, Y. Saad, M.H. Schultz, *Complexity of dense linear system solution on a multiprocessor ring,* Technical Report 349, Computer Science Dept., Yale University, 1984.

[3] S.L. Johnsson, *Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures,* Technical Report YALEU/CSD/RR-361, Dept. of Computer Science, Yale University, September 1985.

[4] J.M. Ortega, R.G. Voigt, *Solution of partial differential equations on vector and parallel computers,* Technical Report 85-1, ICASE, NASA Langley Research Center, 1985.

[5] Y. Saad, M.H. Schultz, *Topological properties of hypercubes,* Technical Report YALEU/DCS/ RR-389, Computer Science Dept., Yale University, 1985.

[6] ———, *Direct parallel methods for solving banded linear systems,* Technical Report YALEU/ DCS/RR-387, Computer Science Dept., Yale University, 1985.

[7] M.H. Schultz, *Multiple Array Processors for Ocean Acoustic Problems,* Technical Report YALEU/DCS/ RR-363, Dept. of Computer Science, Yale University, 1985.

[8] C.L. Seitz, *The Cosmic Cube,* CACM, 28 (1985), pp. 22–33.