Parallel Computing 11 (1989) 131–150
North-Holland

Data communication in parallel architectures *

Youcef SAAD and Martin H. SCHULTZ

Research Institute for Advanced Computer Science, NASA Ames Research Center, Mail Stop 230-5, Moffett Field, CA 94035, U.S.A.

Received July 1986
Revised July 1988

Abstract. In this paper we consider different methods for exchanging data among processors in parallel computers. The most common data exchange operations in parallel numerical methods are examined and different methods for performing them efficiently on each of several ensemble architectures are proposed and analyzed. Our objective is to compare the performance of each particular architecture on each of the data exchange operations.

Keywords. Multiprocessors, parallel architectures, timing estimates for data exchange operations, interprocessor communication.

1. Introduction

Interprocessor communication is often one of the main obstacles to increasing performance of parallel algorithms for multiprocessors. In addition to the actual arithmetic computations, the nodes of a multiprocessor need to exchange data. Many algorithms are communication intensive in that the number of I/O operations in any given node during the execution of the algorithm is not negligible as compared with the number of arithmetic operations. This means that if the I/O channels are slow relative to the CPUs, then the delays incurred by data movements may well dominate the execution time and possibly ruin any hypothetical speedup that would have been expected by considering exclusively the time to perform the arithmetic. This has been illustrated in several papers, see for example [5,6,10].

In this paper, we will examine in detail the problem of exchanging data in multiprocessors and will present several efficient algorithms for performing data exchange tasks. We consider architectures with shared memory, a switch interconnect or a bus interconnect. We also consider loosely coupled nearest neighbor architectures, such as rings, two-dimensional grids, and hypercubes. We study six important exchange operations that arise in various numerical algorithms: data movement between two nodes, broadcasting data from one node to every other node, broadcasting data from every node to every other node, scattering or gathering of data to/from all nodes, and transposition of a matrix.

When programming tasks in parallel machines one may choose to rely entirely on libraries of general purpose software. However, in this case some difficulties, such as memory conflicts in shared memory machines, are never seen by the user. This may result in inefficient programs. For communication intensive problems, it is crucial to carefully organize the data exchange in

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

order to optimize processor utilization. This implies that for parallel multiprocessors, there will be a need for highly optimized subroutine libraries, to carry out the common interprocessor communication tasks, alongside with the more traditional numerical subroutine libraries.

Our goal is to propose and analyze efficient algorithms for communication and to compare the performance of various architectures on the most common data permutation tasks, which would be represented in a communication subroutine library.

In Section 2 we present the six models of architecture we consider and discuss the assumptions and limitations of our models. In Section 3 we consider the data exchange operations one by one, propose algorithms to perform them, and analyze their running times on each of the six architectures.

2. Models of parallel architectures

In this section we introduce the models of architecture we consider in this paper. We examine six different types of architectures that include models for both shared resource machines and nearest neighbor machines.

(a) *The broadcast bus model.* In this model a number of k processors are linked by a global bus which enables broadcasting. Figure 1 is an illustration of such a machine. The broadcast bus is capable of sending data packets from one processor to all other processors in the same amount of time as it would send them to just one processor. Each processor has its own local memory which is sufficiently large to hold the data and code of typical application programs run on the machine. We will assume that the bus is time-shared and that data exchanges are performed exclusively via the bus, although there may exist other means of communication in similar models. We assume that the global bus has sufficient bandwidth to accomodate simultaneous time-shared accesses by k^* processors where $k^* \leqslant k$. In other words the bandwidth b_B of each processor is related to that B of the global bus by $k^* = B/b_B$. To model the times it takes to transfer data, we assume that a packet of m words can be moved between any two nodes in time $\tau_B + m/b_B$, where τ_B is the communication startup and b_B is the bandwidth of the channel from each node to the bus, expressed in words per second.

(b) *The shared memory model.* In the shared memory model the k processors are linked to a global shared memory which they can access with equal speed, see Fig. 2. It is assumed in this model that the global memory is used as a medium of communication between processors. Although it is a common practice that each processor is provided with some local memory , in the form of local cache or sets of data registers, it is assumed here that the data movements considered are performed from global memory and not directly between these local memories. The bandwidth of each processor is b_M, i.e., it takes the time $\tau_M + m/b_M$ to move a packet of m words between any processor and the memory. To allow for efficient access of the memory by several processors, the memory is often divided up in several memory banks. As long as there is no contention to the same memory bank, several processors can read/write simulta-

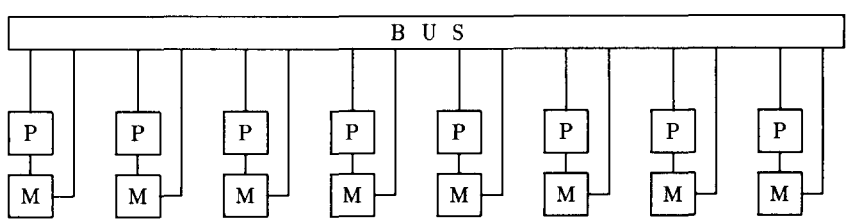

Fig. 1. The broadcast bus model.

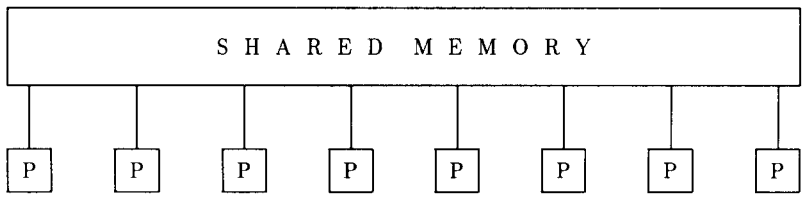

Fig. 2. The shared memory model.

neously. However, it is usually the case that the bandwidth of the global memory does not exceed a multiple k^* of b_M, where $k^* \leqslant k$. This means that at most k^* different processors can access the memory with equal speed b_M in the best possible case where there is no memory conflict. One of the goals for a software developer on this machine is to arrange the data initially so as to minimize memory conflicts. This was examined in detail by Shapiro [15] and more recently by Frailong et al. [4]. Many of the commercially available modern multiprocessors are shared memory machines: CRAY X-MP, CRAY-2, ETA-10 and ALLIANT FX/8.

We will make the assumptions that the data is initially arranged so that there is no memory bank conflict. Since this assumption is not likely to be verified in all practical applications, the timings that we will give are lower bounds that correspond to ideal situations. Note that it would be impossible to give any nonprobabilistic timing estimates without such an assumption.

(c) *The ring architecture.* A multiprocessor ring consists of k processors connected around a ring as is shown in Fig. 3. Each processor can simultaneously write to both neighbors or simultaneously read from one neighbor and write to the other. The time to read/write a packet of m words from/to nearest neighbors is of the form $\tau_R + m/b_R$. Several algorithms for solving dense linear systems on such machines have been considered for such a machine in [8]. Algorithms for rings have also been proposed by Sameh [14].

(d) *Two- and three-dimensional mesh connected arrays.* Two-dimensional and three-dimensional arrays are popular among partial differential equations specialists because they offer a simple way of mapping regular finite difference grids into the multiprocessor grid. A square two-dimensional grid consists of an array of $\sqrt{k} \times \sqrt{k}$ processors connected as shown in Fig. 4. In this paper we will only consider two-dimensional arrays although many of the algorithms to be described extend easily to three-dimensional grids. Moreover, for convenience we assume that the processors on each side of the grid are connected to those on the opposite side. We assume these wrap around connections in order to yield more homogeneous complexity results. These connections are not essential for the algorithms themselves. Our assumptions are

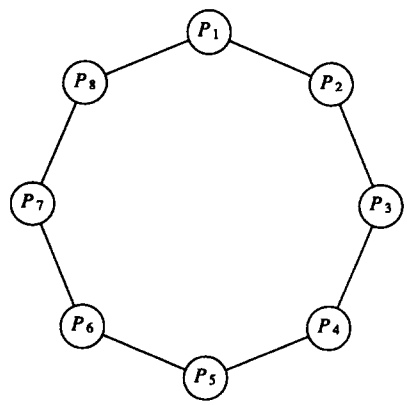

Fig. 3. A processor ring consisting of eight processors.

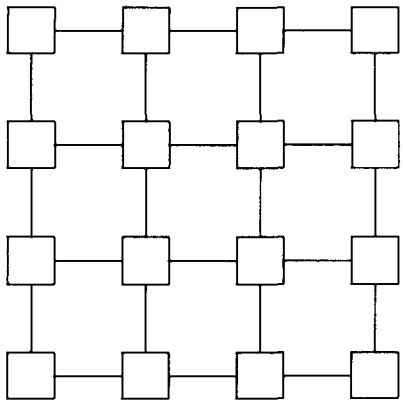

Fig. 4. A 4 × 4 multiprocessor grid.

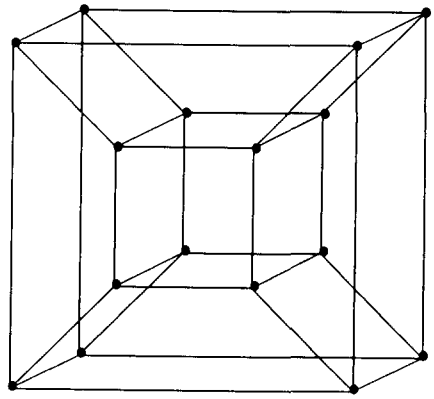

Fig. 5. Three-dimensional view of a hypercube of dimension 4.

identical with those of the ring model but one processor is now able to simultaneously communicate with its four neighbors. The time required to send m words from one processor to one of its neighbors is given by

$$\tau_G + m/b_G, \tag{2.1}$$

where τ_G is the grid latency time, and b_G is the bandwidth of each link.

(e) *Hypercubes.* We refer to any loosely coupled parallel processor based on the binary n-cube network as a hypercube. An n-dimensional hypercube consists of $k = 2^n$ identical processors, numbered from 0 to $2^n - 1$ and connected in such a way that there is a link between two processors if and only if the binary representations of their numbers differ by one and only one bit. Thus any node has exactly n neighbors. For example, the 8 nodes of a 3-dimensional cube ($n = 3$) can be represented as the vertices of a three dimensional cube. Figure 5 illustrates a four dimensional cube. We assume that, it takes the time

$$\tau_H + m/b_H, \tag{2.2}$$

to transfer m words from one processor to any number of its n neighbors. Moreover, we assume as before that communication in all of the n directions can take place simultaneously: all the channels of a node can either read or write, but not both, simultaneously from or to a neighbor. This assumption is important because it implies that the total communication speed from a node is proportional to its degree. Without it the hypercube would be communication inefficient in that it would possess hardware that is not utilized to its full capacity.

Some properties of the hypercube topology have been developed in [11] and communication problems in the hypercube have been examined in detail in [13]. The same algorithms will be presented in this paper in condensed form and the reader is referred to the above report for details.

(f) *The switch connected model.* In switch connected models, k processors are linked to k other processors in a one-to-one fashion, by means of the switching network, see Fig. 6. We assume that the switch is capable of connecting k inputs to k outputs in a one-to-one fashion, thus realizing the permutation

$$P_1, P_2, \ldots P_k \rightarrow P_{i_1}, P_{i_2}, \ldots, P_{i_k}.$$

Once the switch is configured, communication between connected processors can take place simultaneously. Ideally, we would like to consider cross-bar switches which can realize an arbitrary permutation but these are too expensive to build when the number of processors is

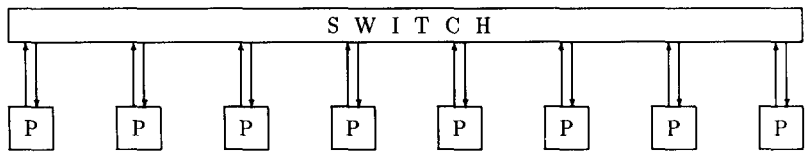

Fig. 6. A switch connected machine.

large. There are switching networks which can realize all permutations in a number of $O(\log_2 k)$ stages thus offering a seemingly perfect alternative to cross-bar switches. However, there is one important difficulty with this type of switches which is that they require a time of order $O(k \log_2 k)$ to be set, or time $O(\log_2^2 k)$ using a parallel algorithm. The most popular switching networks are those that realize an important subset of permutations, usually including all permutations that are important in numerical computations, in $\log_2 k$ stages. These include the Omega network [9], the inverse Omega network, the indirect binary n-cube, and the data manipulator network. For an exhaustive description of these networks the reader is referred to [16]. Most of them are topologically equivalent [2]. In these networks, the data movements are determined by the data packet itself. One of the most popular is the Omega network of Lawrie [9]. The time to send a packet of m words simultaneously from any group of the processors P_1, $P_2, \ldots, P_k$ to their destinations among the processors $P_{i_1}, P_{i_2}, \ldots, P_{i_k}$, can be modeled by the formula $\tau_S + m/b_S$, where τ_S is the setup time which includes the switch delays and b_S is the bandwidth of each channel from the switch to a node. Note that for machines where each word, or block of words, is transmitted with a new startup, the formula is still valid with τ_S replaced by 0 and b_S replaced by some actual speed that takes into account these startups.

Among the contemporary machines based on switching networks are the Staran [1,16] the BBN butterfly [3], and the NYU Ultracomputer [7].

3. Communication algorithms

In this section we will consider the following data exchange operations:

(1) Moving data from one processor to another. This represents the simplest data transfer operation.

(2) Moving the same data packet from one processor to all others. We will refer to this as a broadcast operation.

(3) Moving a data packet from each processor to every other processor. This is in effect a broadcast operation (2) from each node.

(4) Scattering (gathering) data from (in) one processor to (from) all others. In the scatter operation, a node sends a packet to every other processor. These packets, although different, are ideally all of the same size. The gather operation is the dual operation: the node receives a packet of (ideally) equal size from each of the other nodes.

(5) Multiscattering or multigathering of data. The multiscatter operation consists of a scatter from every node. Multigather is defined similarly. Both operations will also be referred to as transposition of data, as they essentially amount to transposing a matrix which is distributed by rows or columns among the processors.

The difference between the broadcast (2) and the scatter (4) is that in the scatter operations a different data set is sent to every processor. The gather and scatter operations are very similar in nature: an algorithm for scatter can be derived from an algorithm for gathering simply reversing the data paths and vice-versa. For this reason we will only consider scatter operations in the rest of the paper.

3.1. Moving data from one node to another

We consider the problem of moving N words from one node A to another node B.

(a) *Broadcast bus.* Using the broadcast bus, it takes the time

$$T_{1,\mathrm{B}} = \frac{N}{b_{\mathrm{B}}} + \tau_{\mathrm{B}}. \tag{3.1}$$

(b) *Shared memory.* To move N elements from node A to node B we need to write the N elements to the global memory, and then read them from memory to node B. Assuming there is no memory conflict, the total time would be

$$T_{1,\mathrm{M}} = 2\left(\frac{N}{b_{\mathrm{M}}} + \tau_{\mathrm{M}}\right). \tag{3.2}$$

(c) *Ring.* There are several approaches. The simplest (naive) one consists of hopping the data from A to B along the shortest path between these two nodes. This requires a total of $dist(A, B)$ steps each of which is a transfer of N elements between nearest neighbors. The total time is

$$T^{N}_{1,\mathrm{R}} = i\left(\frac{\tau_{\mathrm{R}} + N}{b_{\mathrm{R}}}\right), \tag{3.3}$$

where $i \equiv dist(A, B)$.

In the above naive scheme the links are not used in an optimal manner, since they are used one at a time and parallelism in the data transfer is not exploited. As was suggested in [8,12] a better alternative is to split the data into ν different packets of equal size and pipeline the data transfer from $P_0 \equiv A$ to $P_i \equiv B$ along the shortest path $P_0, P_1, P_2, \ldots, P_i$ from A to B. Note that in a ring there are two paths between any two given nodes and that the shortest one has a length $i \leqslant \lfloor \frac{1}{2}k \rfloor$. Thus, at step 1 the first packet is sent from P_0 to P_1. At step 2, while sending the first packet to P_2, P_1 receives the second packet from P_0. Generally, at step j, the first packet reaches P_j from P_{j-1} while the second packet follows from P_{j-2} to P_{j-1}, etc... The first packet reaches P_i at the ith step and $\nu - 1$ more steps are needed for the remaining packets to reach P_i, resulting in a total of $\nu - 1 + i$ steps and a time of

$$t(\nu) = (\nu - 1 + i)\left(\tau_{\mathrm{R}} + \frac{N}{\nu b_{\mathrm{R}}}\right).$$

The above time is minimized for the optimal number of packets equal to

$$\nu_{\mathrm{opt}}(N, i) = \sqrt{\frac{(i-1)N}{b_{\mathrm{R}}\tau_{\mathrm{R}}}}, \tag{3.4}$$

and the corresponding optimal time is

$$T^{P}_{1,\mathrm{R}} = \left(\sqrt{\frac{N}{b_{\mathrm{R}}}} + \sqrt{(i-1)\tau_{\mathrm{R}}}\right)^2, \tag{3.5}$$

where $i \equiv dist(A, B) \leqslant \lfloor \frac{1}{2}k \rfloor$.

It is important to observe that the maximum distance i between two nodes in a ring is bounded by $\frac{1}{2}k$. Since we must have $1 \leqslant \nu \leqslant N$, formula (3.4) is valid only when

$$\frac{1}{N(i-1)} \leqslant \frac{1}{b_{\mathrm{R}}\tau_{\mathrm{R}}} \leqslant \frac{N}{i-1}.$$

This will always be the case if N is large enough with respect to the ratio $1/b_{\mathrm{R}}\tau_{\mathrm{R}}$, which is

usually small. However, in case $1/[N(i-1)] > 1/(b_R\tau_R)$, the optimal number of packets is $\nu = 1$ and the above timing formula becomes $T^P_{1,R} = i(N/b_R + \tau_R)$. When $1/(b_R\tau_R) > N/(i-1)$, the optimal number of packets is $\nu = m$ and the optimal time becomes $T^P_{1,R} = (N + i - 1)(1/b_R + \tau_R)$.

Formula (3.5) expresses the time in which N words can be moved *along a chosen path of length i* between two processors. Since there are two different paths between any two nodes A and B, an improvement over the above scheme is to divide the data in two sets and send each set along one of the two paths. Note that now we can only guarantee that both paths are of length not exceeding $k-1$. Assuming that we divide the set into two equal parts, and that we pipeline the data transfer in the same way as above along the two paths the above time becomes

$$T^{P2}_{1,R} = \left(\sqrt{\frac{N}{2b_R}} + \sqrt{(i-1)\tau_R}\right)^2. \tag{3.6}$$

where $i \leqslant k-1$.

The above time is about half that of the one-way pipelining time in case the amount of data, i.e., N, is very large. Note that sending half the data in each of the two routes may not the best that can be done. It can be shown that there is an optimal ratio of the size of the two data sets over that of the original one so as to minimize the total time. The details are omitted as the resulting formulas are complicated.

(d) *Two-dimensional grids.* If we choose to send the data along a single path, then the only difference between the situation of a grid and that of the ring is the shorter distance of the path which, assuming $\sqrt{k}$ is even, is now such that

$$i \equiv dist(A, B) \leqslant 2\lfloor \tfrac{1}{2}\sqrt{k} \rfloor = \sqrt{k}. \tag{3.7}$$

This is because it takes at most $\lfloor \frac{1}{2}\sqrt{k} \rfloor$ to reach the processor of the grid which is on the same horizontal level as B and, likewise, at most $\lfloor \frac{1}{2}\sqrt{k} \rfloor$ to reach B from there. As a result a formula similar to (3.5) is

$$T^P_{1,G} = \left(\sqrt{\frac{N}{b_G}} + \sqrt{(i-1)\tau_G}\right)^2, \tag{3.8}$$

where i satisfies (3.7).

Just as was done for the ring, we can exploit alternative paths between two nodes. In the present situation, we can use 4 paths simultaneously to travel from A to B. One restriction on these paths is that they do not use common edges, i.e., that they be edge-disjoint.

Assume at first that A and B do not lie on the same line of the processor grid. Then it is easily seen that there are two obvious parallel paths between these two nodes. The first one starts by traveling in the vertical direction until reaching the horizontal line where B lies and then travels in the horizontal direction to reach B. The second starts by traveling in the horizontal direction and then in the vertical direction. Both paths are optimal in the sense that their length is exactly $dist(A, B)$. Two additional paths can be found by going around the rectangle formed by the first two paths. Each of the additional paths is of length at least $i + 4$. But as is easily seen this can be improved slightly by having 4 paths of length $i + 2$ each. The first two of these four paths are shown in Fig. 7. The other two, not shown in the figure for clarity, are obtained by taking the reflections of these paths with respect to the axis A, B.

Moreover, it can be shown that this cannot be improved, i.e., that it is not possible to find four paths all of which are of length less than $+2$ if we want to minimize the maximum length of all 4 paths. This is an immediate consequence of the following lemma.

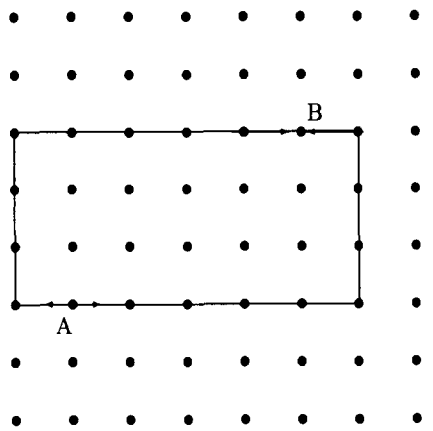

Fig. 7. Edge-disjoint paths in a two-dimensional grid: the general case.

Lemma 3.1. *If l_1, l_2, l_3, l_4 are the lengths of four edge-disjoint paths between two nonaligned nodes A and B, then*

$$\sum_{i=1}^{4} l_i \geqslant 4[dist(A, B) + 2]. \tag{3.9}$$

Proof. Consider the ith path and let A_i be the first vertex visited after leaving node A and B_i the last vertex visited before the final vertex B. Observe that the A_i's, $i = 1,\ldots,4$ must be the four neighbors of A. Similarly, the B_i's, $i = 1,\ldots,4$ must be the four neighbors of B. It is clear that

$$l_i \geqslant dist(A_i, B_i) + 2, \quad i = 1,\ldots,4,$$

and therefore,

$$\sum_{i=1}^{4} l_i \geqslant \sum_{i=1}^{4} dist(A_i, B_i) + 8, \quad i = 1,\ldots,4. \tag{3.10}$$

Let us now choose a coordinate system for the vertices and denote by M^x, M^y the coordinates of a vertex M with respect to this system. It is assumed without loss of generality that $A^x \leqslant B^x$ and $A^y \leqslant B^y$. Since $dist(A_i, B_i) = |B_i^x - A_i^x| + |B_i^y - A_i^y|$, we have

$$\begin{aligned}\sum_{i=1}^{4} dist(A_i, B_i) &= \sum_{i=1}^{4} \left[|B_i^x - A_i^x| + |B_i^y - A_i^y| \right] \\ &\geqslant \sum_{i=1}^{4} \left[(B_i^x - A_i^x) + (B_i^y - A_i^y) \right] = 4\left[(B^x - A^x) + (B^y - A^y) \right] \\ &= 4\ dist(A, B).\end{aligned}$$

The proof follows by replacing the above inequality in (3.10). □

A consequence of the lemma is that there are only two possible ways in which the sum of the 4 path lengths can be minimized: either two paths of length $dist(A, B)$ and two of length $dist(A, B) + 4$ or four paths of length $dist(A, B) + 2$. The second possibility minimizes the maximum path length.

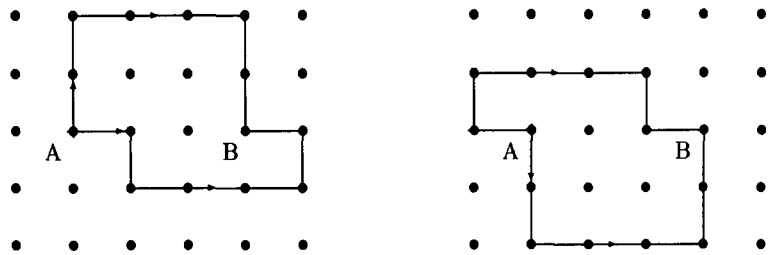

Fig. 8. Four edge-disjoint paths in the aligned case.

In case A and B are on the same line, two of the paths collapse into each other and we need to define the four paths differently. One solution is given in Fig. 8: there are four paths of length $i + 4$ each.

Finally, in the ultimately particular case when the two vertices are neighbors, the previous solution fails again and one can find 4 paths between A and B which are of length at most $i + 6$, more precisely: one of length one $(= i)$, one of length 3 $(= i + 2)$, one of length 5 $(= i + 4)$ and finally one of length 7 $(= i + 6)$. Thus we have shown the following result.

Proposition 3.2. *There are four edge-disjoint paths between any two nodes A, B of a 2-dimensional grid (with wrap around). Moreover, in the case where A and B are not aligned in the grid, these paths can be chosen to be of length $dist(A, B) + 2$ each. If A and B are aligned (in the horizontal or vertical direction) but not neighbors, then the four paths can be chosen to be of length $dist(A, B) + 4$. Finally if A and B are neighbors, then the four paths can be chosen to be of length at most $dist(A, B) + 6 = 7$.*

It follows from Proposition 3.2 that data can be routed through 4 parallel paths between two nodes of a two-dimensional grid, with the maximum path length of $dist(A, B) + 6$. Therefore the corresponding time is bounded as follows:

$$T^P_{1,G} = \left(\sqrt{\frac{N}{4b_G}} + \sqrt{(i-1)\tau_G} \right)^2, \tag{3.11}$$

where

$$i \leqslant dist(A, B) + 6 \leqslant \sqrt{k} + 6. \tag{3.12}$$

We should emphasize that the above formula is not optimal since it uses the bound related to the particular case where $dist(A, B) = 1$ which can be handled in a more efficient way by sending more information over the shortest path of length one.

(e) *Hypercubes.* The problem of moving data among the nodes of a hypercube was examined in detail in [13]. Let A and B be any two nodes of an 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 $A = a_1 a_2 .. a_n$ successively into those of $B \equiv b_1 b_2 .. b_n$, starting from the left [11]. 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. This path length cannot exceed n, the dimension of the cube:

$$dist(A, B) \leqslant n = \log_2(k). \tag{3.13}$$

If only one path is used to move data from a node to another, then the data transfer can clearly be completed in time

$$T_{1,\mathrm{H}}^{P} = \left(\sqrt{\frac{N}{b_{\mathrm{H}}}} + \sqrt{(i-1)\tau_{\mathrm{H}}} \right)^2, \tag{3.14}$$

where i satisfies (3.13).

As in the previous nearest neighbor architectures we now raise the question, how many parallel paths between A and B can be found? In the above path between A and B there is no reason why we must start by correcting the leftmost, i.e., the first, differing bit. More generally we might start correcting the jth bit (where $j \leqslant i$) then the $(j+1)$st bit and so forth until the nth bit is reached, after which we correct in turn bits 1, 2,...$(j-1)$. We can thus define i different paths. It was shown in [11] 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 [11] 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 $i + 2$. The following proposition summarizes these results.

Proposition 3.3. *Let A and B by any two nodes of an n-cube and let $i = H(A, B)$ be 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 $i + 2$.*

We now wish to estimate the time it takes to transfer data between any two nodes. Assume that the design of each node is such that it is capable of sending a different message to each of its n adjacent nodes at once, i.e., that it is capable of overlapping communication in 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. Moreover, 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:

$$t_{1,\mathrm{H}}^{P} = \left(\sqrt{\frac{N}{(\log_2 k) b_{\mathrm{H}}}} + \sqrt{(i-1)\tau_{\mathrm{H}}} \right)^2,$$

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 the number of edges per node, is reflected by a division of the total data length N by n.

(f) *Switch.* In this model a simple shift to the required connection followed by the transfer will achieve the data movement in the approximate time:

$$T_{1,\mathrm{S}} = \tau_{\mathrm{S}} + \frac{N}{b_{\mathrm{S}}}.$$

3.2. Broadcasting

(a) *Broadcast bus.* The time for broadcasting a message of length N is

$$T_{2,\mathrm{B}} = \frac{N}{b_{\mathrm{B}}} + \tau_{\mathrm{B}}.$$

(b) *Shared memory.* To broadcast data from one processor to all others in a shared memory multiprocessor, one may write the data into memory at the cost of $N/b_{\mathrm{M}} + \tau_{\mathrm{M}}$ and then, in $\max(1, k/k^*)$ times, have groups of k^* processors read it up simultaneously from memory at

the cost of $(N/b_M + \tau_M)\max(1, k/k^*)$. However, we must be careful that when all the processors will read simultaneously from the same location, this will result in memory conflict. Assuming that the data is spread in sufficiently many different memory banks and that there will be no bank contentions the total time comes to

$$T_{2,M} = \left(\frac{N}{b_M} + \tau_M\right)\left(1 + \max\left(1, \frac{k}{k^*}\right)\right). \tag{3.15}$$

(c) *Ring*. One way to send a data packet of size N to all processors in a ring is to pipeline the data around in the maximum length path. According to Section 3.1 the time for this is given by (3.5) in which i is replaced by the maximum length $k-1$, i.e.,

$$T^1_{2,R} = \left(\sqrt{\frac{N}{b_R}} + \sqrt{(k-2)\tau_R}\right)^2. \tag{3.16}$$

A better way is to broadcast the data to the processors on both sides of the transmitting node. Thus the broadcasting node will have to send the same amount of data both ways but this time the farthest node to reach in each way is only at distance $\lfloor \frac{1}{2}k \rfloor$ away from it. Hence the time,

$$T^2_{2,R} = \left(\sqrt{\frac{N}{b_R}} + \sqrt{(i-1)\tau_R}\right)^2, \tag{3.17}$$

where $i = \lfloor \frac{1}{2}k \rfloor$.

(d) *Grid*. An analogue of the second method used for the ring can be developed. The broadcasting node starts by sending the same data packet to its four neighbors which in turn relay it to their neighbors and so on while receiving the second packet and so forth. During the pipelined transfer, each receiving node forwards the received packet to some of its neighbors. In order to avoid unnecessary duplication of the data in the receiving processors, some order is necessary in the transfer. Let us assume that the processors are numbered by the double index (i, j), where i is the number of the row to which the processor belongs and j is the column number, relative to the transmitting processor, which is numbered (0, 0). The indices can assume negative values: the $\frac{1}{2}\sqrt{k}$ columns at the left of (0, 0) are number decreasingly $-1, -2,...$ We make a similar convention when numbering the rows. Then after the first step, processors $(-1, 0)$, $(0, 1)$, $(1, 0)$, and $(0, -1)$ will have received their first packet. More generally, after the ith step of the algorithm, all processors labeled (h, j) with $|h+j| = i$ have just received the first packet. At the next step we proceed as follows. Processors $(i, 0)$ and $(-i, 0)$, i.e., those located at the north and south of the origin (0, 0), will transfer the first packet to the north and south respectively, i.e., to processors $(0, i+1)$, and $(0, -i-1)$ respectively while receiving the second packet from the processors $(0, i-1)$ and $(0, -i+1)$ respectively. Meanwhile, processors labeled (h, j) with $|h+j| = i$ and $h > 0$ will transmit the first packet to their eastern neighbors, i.e., to processors $(h+1, j)$ while receiving the second packet from their western neighbors, i.e. from $(h-1, j)$. On the western side, processors labeled (h, j) with $|h+j| = i$ and $h < 0$ will transmit the first packet to their western neighbors, i.e., to processors $(h-1, j)$ while receiving the second packet from their western neighbors, i.e. from $(h+1, j)$. This is illustrated in Fig. 9. Pipelining inside the square of nodes (h, j) such that $|h+j| = i$ is performed in a straightforward way. For all the data to reach all processors, we need the same as if we pipelined it along a single path from node (0, 0) to the farthest node from it which is at the distance $2\lfloor \frac{1}{2}\sqrt{k} \rfloor$. According to Section 1, this takes the time

$$T_{2,G} = \left(\sqrt{\frac{N}{b_G}} + \sqrt{(i-1)\tau_G}\right)^2, \tag{3.18}$$

where $i = 2\lfloor \sqrt{\frac{1}{4}k} \rfloor$.

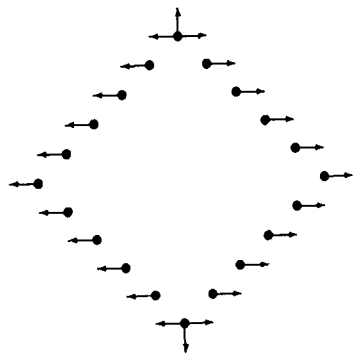

Fig. 9. Movement of first packet in the 6th step of the grid broadcasting.

It is important to observe that the vertical links of the processor grid are idle most of the time. This implies that the algorithm is not optimal and raises the question of whether it is possible to improve its performance. The idea of the version proposed next is to try to optimize the above algorithm by sending half of the data vertically and the other half horizontally.

Let us split the data in two equal parts and call these two parts D_x and D_y. Each part is further plit into smaller packets for pipelining. In the very first step the node (0, 0) simultaneously sends the first D_y packet to the east and west neighbors (1, 0), $(-1, 0)$ and the first D_x packet to the north and south neighbors (0, 1), $(0, -1)$. At the ith step, with $i > 1$, processors $(0, j)$, $j \leqslant i$, send the D_x packets received in the previous step to processors to their north, east and west neighbors, i.e., to processors $(0, j+1)$, $(1, j)$ and $(-1, j)$, while processors $(0, -j)$ will send these same packets to their south, east and west neighbors, i.e., to processors $(0, -j-1)$, $(1, -j)$ $(-1, -j)$ respectively. Similarly, processors $(j, 0)$, $j \leqslant i$, send the D_y packets received in the previous step to processors to their east, north and south neighbors, i.e., to processors $(j+1, 0)$, $(j, 1)$ and $(j, -1)$, while processors $(-j, 0)$ will send these same packets to their west, north and south neighbors, i.e., to processors $(-j-1, 0)$, $(-j, 1)$ $(-j, -1)$ respectively. A general processor (h, j) with $h + j \leqslant i$ and $h \neq 0$, $j \neq 0$ will send the D_x packets horizontally and outward from (0, 0), i.e., to processors $(h + sign(h), j)$ and the D_y packets vertically and outward from (0, 0), i.e., to the neighbors $(h, j + sign(j))$. This is illustrated in Fig. 10, for the case $i = 6$

An important detail remains to be addressed: so far processors numbered $(0, j)$ have not yet received a D_y packet and processors $(j, 0)$ have not received a D_x packet. All other processors

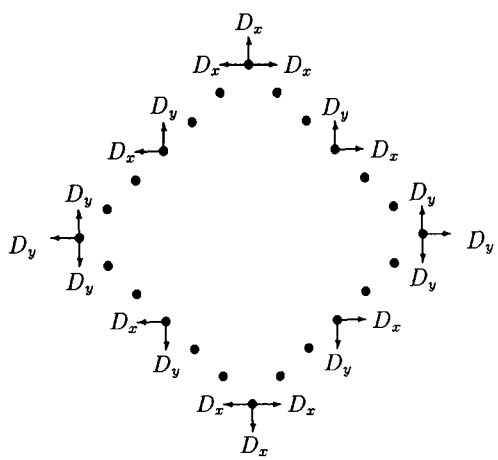

Fig. 10. Movement of first packet in the 6th step of the optimal grid broadcasting algorithm.

will get all the desired packets after $\nu_{\text{opt}} + 2\sqrt{k}$ steps. An easy remedy to this is therefore to have the neighbors of these particular nodes send the missing data to them. But a more attractive solution is to observe that after the last pair of D_x, D_y leaves from node (0, 0) there are still $2\sqrt{k}$ pipelining steps left which can be used to pipeline the D_x data to the processors $(j, 0)$ and the D_y data to processors $(0, j)$. An important observation is that these two additional restricted broadcast operations in the horizontal and vertical lines will end at the same time as the rest of the broadcast operation. As a result the time required by this optimal algorithm is

$$T^2_{2,\text{G}} = \left(\sqrt{\frac{N}{2b_{\text{G}}}} + \sqrt{(i-1)\tau_{\text{G}}} \right)^2, \tag{3.19}$$

where $i = 2\lfloor \sqrt{\frac{1}{4}k} \rfloor$.

A third method, which is somewhat simpler, is to pipeline the data along the vertical line on which the processor lies, i.e., to all processors labeled $(i, 0)$ and then each processor $(i, 0)$ broadcasts the data in its horizontal direction, i.e., to all processors $(i, *)$. This requires the time

$$T^3_{2,\text{G}} = 2\left(\sqrt{\frac{N}{b_{\text{G}}}} + \sqrt{(i-1)\tau_{\text{G}}} \right)^2, \tag{3.20}$$

where $i = \lfloor \frac{1}{2}\sqrt{k} \rfloor$. This is always larger than the time (3.18).

(e) *Hypercube*. Broadcasting data on hypercubes was considered in detail in [11] where we proposed an algorithm based on a spanning tree of the hypercube. The optimal time, using pipelining, is approximately

$$T_{2,\text{H}} \approx \left(\sqrt{\frac{N}{b_{\text{H}}}} + \sqrt{(\log_2 k - 1)\tau_{\text{H}}} \right)^2. \tag{3.21}$$

(f) *Switch*. Many switching networks provide for a broadcasting mechanism by making each of the 2×2 crossbar switches capable of broadcasting from one input to two outputs. Thus the Omega network is able to broadcast information from any input to all outputs.

Others do not offer this capability and broadcasting must be done by software. For the latter case we will propose several solutions of improving quality. The first method consists of embedding a linear array into the switch and then using one of the algorithms described for the ring. The permutation needed to achieve the embedding of a ring into the switch is the following $\Pi(j) = (j \bmod k) + 1$.

The time required to complete the algorithm using pipelining along one direction only of the ring as given by (3.16) becomes

$$T^1_{2,\text{S}} = \left(\sqrt{\frac{N}{b_{\text{S}}}} + \sqrt{(k-2)\tau_{\text{S}}} \right)^2. \tag{3.22}$$

Note that we do not use the two way pipelining in the ring here, i.e., we only travel in one way around the ring. The reason is that the ring to switch embedding described above does not allow a given node to send data to more than one processor, which implies that the transmitting node cannot move data in two directions at once.

A second algorithm is to first send the data to one other node. Then the two nodes send in parallel their data to two other nodes and so on. After i steps, 2^i nodes will have received the data. We therefore need a total of $\log_2(k)$ steps to complete the broadcast and a total time of

$$T^2_{2,\text{S}} = \log_2(k)\left(\frac{N}{b_{\text{S}}} + \tau_{\text{S}} \right). \tag{3.23}$$

A third algorithm consists of emulating a binary tree with the switch and then pipelining the data in the tree. Assume, conceptually, that the k nodes are the nodes of a binary tree. We now consider two partial permutations of the switch. The first maps every parent in the tree into its left child and the second maps every parent to its right child. We can now pipeline the data into all processors by alternating left and right transfers: in each step of the transfer a node at some level i receives a packet and then transmits it to its left child first then to its right child. If the data is split into ν packets of equal size the first packet reaches the leaves at step $\log_2 k - 1$, and then we need another $\nu - 1$ steps to have all packets in every node. The total time required for this data transfer is therefore

$$(\log_2 k - 1 + \nu - 1) \cdot 2 \cdot \left(\frac{N}{\nu b_S} + \tau_S \right)$$

which is optimized for

$$\nu_{opt} = \sqrt{\frac{(\log_2(k) - s)N}{b_S \tau_S}}, \tag{3.24}$$

and yields the optimal time:

$$T^3_{2,S} = 2\left(\sqrt{\frac{N}{b_S}} + \sqrt{(\log_2(k) - 2)\tau_S} \right)^2. \tag{3.25}$$

3.3. Total exchange

We assume in the following discussion that a data set of N/k words must be moved from every processor to every other processor. This sort of data permutation is of importance in many numerical algorithms, see [13] for a few illustrations.

(a) *Broadcast bus.* We must simply broadcast k times a set of size N/k in time

$$T_{3,B} = k\left(\frac{N}{k b_B} + \tau_B \right) = \frac{N}{b_B} + k\tau_B. \tag{3.26}$$

(b) *Shared memory.* We can map all the data into memory and then read it back to all processors. In the write part each processor writes N/k words into memory in time $(N/(kb_M) + \tau_M)\ max(1, k/k^*)$ and in the read part every processor reads $N - N/k$ words in time $((N - N/k)/b_M + \tau_M)\ max(1, k/k^*)$. Therefore,

$$T_{3,M} = max\left(1, \frac{k}{k^*}\right)\left(\frac{N}{b_M} + 2\tau_M \right). \tag{3.27}$$

(c) *Ring.* An effective method consists of rotating the data in a round-robin or daisy chain fashion. In the first step every processor sends its data to the right. In all subsequent steps, every processor forwards to its right the data that it has just received in the previous step. After $k - 1$ such rotations, every processor will wind up with the data from every other processor. The total time required is

$$T_{3,R} = (k - 1)\left(\frac{N}{k b_R} + \tau_R \right). \tag{3.28}$$

(d) *Grid.* There are several methods which we will describe in increasing order of efficiency. The simplest method of all is to broadcast from every node in turn. According to Section 3.2,

this sequential broadcasting algorithm takes time

$$T_{3,G}^{SB} = \left(\sqrt{\frac{N}{kb_G}} + \sqrt{(\sqrt{k}-1)\tau_G} \right)^2, \tag{3.29}$$

in the case where $\sqrt{k}$ is even.

A second method, which we refer to as daisy chaining, is based on embedding a ring into the grid and then using the algorithm for the ring. It is possible to embed a ring into a grid when the number of grid points is even. The length of the ring is then equal to the number of nodes, i.e., to k. Hence,

$$T_{3,G}^{DC} = (k-1)\left(\frac{N}{kb_G} + \tau_G \right). \tag{3.30}$$

Clearly, daisy chaining is suboptimal because it uses a very small portion of the total bandwidth of the grid of processors.

Our third method is a combination of daisy chaining in the vertical and horizontal directions of the grid. In the first pass, consisting of $\sqrt{k}-1$ steps, all nodes collect the data items of the other nodes on their vertical line by the daisy chaining algorithm of the ring. Since each vertical line forms a ring of $\sqrt{k}$ processors, this first pass requires a total of $(\sqrt{k}-1)(N/(kb_G)+\tau_G)$. In the second pass, each node will perform a daisy chaining in the horizontal direction to get the data collected from the first pass. The data packet moved around each horizontal line has now swelled to $\sqrt{k}N/k = N/\sqrt{k}$ and therefore the total time for the second pass is $(\sqrt{k}-1)(N/(\sqrt{k}\,b_G)+\tau_G)$, leading to a total time of

$$T_{3,G}^{AD} = (\sqrt{k}-1)\left(\frac{N}{\sqrt{k}\,b_G}\left(1+\frac{1}{\sqrt{k}}\right) + 2\tau_G \right). \tag{3.31}$$

Notice that for large k this is approximately

$$T_{3,G}^{AD} \approx \left(\frac{N}{b_G} + 2\sqrt{k}\,\tau_G \right). \tag{3.32}$$

Observe that the latency time is divided by a factor of $\frac{1}{2}\sqrt{k}$ as compared with the time of the daisy chaining algorithm.

(e) *Hypercube.* Several methods for exchanging data from every node to every other node have been seen in [13]. The most effective one described there is a total exchange algorithm which essentially keeps all the links busy exchanging data all the time. The corresponding time for large k is the form

$$T_{3,H} \approx 2(\log_2 k)\tau_H + \frac{2N}{(\log_2 k)b_H}. \tag{3.33}$$

(f) *Switch.* A first method would be to emulate the ring (see Section 3.2) and use the daisy chaining algorithm of the rings. The corresponding time would be

$$T_{3,S} = (k-1)\left(\frac{N}{kb_S} + \tau_S \right). \tag{3.34}$$

A better time can be achieved by a simple variation of the daisy chaining scheme. For example, a permutation used in emulating the ring is, $\Pi(j) = (j \bmod k) + 1$. In the first step of the daisy chaining, each processor has its data plus the data of its right neighbor. What we would like at the next step is that any given processor collects the two data items accumulated two processors away (at the right) from it. Thus, processor 1 will receive from processor number 3 the data of processor 3 and that of its right neighbor, which processor 3 has obtained

in step 1. At step 3, we should get the 4 data items accumulated in the processor at distance 4 from it and so on. The algorithm can be sketched as follows:

For $i = 1, 2, \ldots \log_2 k - 1$ do

Every processor j reads the data collected in processor number $[(j - 1 + 2^i) \bmod k] + 1$.

This main loop of the above algorithm can be performed in parallel thanks to the switch mapping used at the ith step $\Pi(j) = [(j - 1 + 2^i) \bmod k] + 1$. The data to be moved doubles at every step, and the number of steps is $\log_2(k)$. This leads to the total time

$$T^2_{3,S} \approx \frac{N}{b_S} + \log_2(k)\tau_S. \tag{3.35}$$

We should point out that this algorithm is a simple adaptation of a similar algorithm called the alternating direction exchange algorithm for hypercubes in [13].

3.4. Scattering and gathering

We consider the problem of scattering k vectors each of size N/k from one processor to all others.

(a) *Broadcast bus.* It suffices to send the data packets successively, in time

$$T_{4,B} = k\left(\frac{N}{kb_B} + \tau_B\right) = k\tau_B + \frac{N}{b_B}. \tag{3.36}$$

(b) *Shared memory.* To scatter k different vectors of length N/k each from one processor to all others in a shared memory multiprocessor, we may write the data into memory at the cost of $N/b_M + \tau_M$ and then have all processors read it up simultaneously from memory at the cost of $(N/b_M + \tau_M)$ $max(1, k/k^*)$. Again we should be aware of memory conflict. Assuming no memory conflict the total time comes to approximately

$$T_{4,M} = \left(\frac{N}{b_M} + \tau_M\right)\left(max\left(1, \frac{k}{k^*}\right) + 1\right). \tag{3.37}$$

(c) *Ring.* To scatter data around a ring we will pipeline it around in both directions in parallel, the first packet being the packet sent counter-clockwise to processor $\lceil \frac{1}{2}k \rceil$ and the second being sent clockwise to processor $\lceil \frac{1}{2}k \rceil + 1$ and so on. The data transfer will be completed as soon as the first packet reaches its destination, i.e., in time

$$T_{4,R} \approx \left(\frac{N}{2b_R} + \lceil \tfrac{1}{2}k \rceil \tau_R\right). \tag{3.38}$$

(d) *Grid.* To scatter data from node (0, 0) in the grid, we can proceed as for the broadcast operation. We can do a scatter in the horizontal direction, where we send to processor $(i, 0)$ all packets that are destined to all processors with indices (i, j). In the second phase all processors $(i, 0)$ scatter in the vertical direction those data packets they received in the first phase. According to the previous paragraph, the time for phase 1 is about

$$\frac{N}{2b_R} + \sqrt{\lceil \tfrac{1}{2}k \rceil}\,\tau_G.$$

The second phase will take the time

$$\tfrac{1}{2}\frac{N}{\sqrt{k}\,b_G} + \sqrt{\lceil \tfrac{1}{2}k \rceil}\,\tau_G,$$

and the total is

$$\frac{N}{2b_G}(1 + k^{-1/2}) + 2\sqrt{\lceil \tfrac{1}{2}k \rceil}\,\tau_G \approx \frac{N}{2b_G} + \sqrt{\tfrac{1}{2}k}\,\tau_G. \tag{3.39}$$

(e) *Hypercube*. Scattering algorithms can be easily derived by analogy with broadcasting algorithms. The difference between the two is that the first sends the same data to all processors while the second sends a different data packet from a particular node (say node $O = 0^n$) to all other processors. Assume that we want to scatter the vectors $v_0, v_1, \ldots, v_{k-1}$ to the nodes $0, 1, \ldots, k-1$ respectively. In [13] we have derived the following algorithm based on this observation.

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 ith step of the algorithm, every node A with a label of the form $0^{n-j+1}a_j$, contains the subvectors with labels xa_j, x being an arbitrary $(n-j+1)$-binary number. Each node A keeps the subvector that is destinated to it and forwards those subvectors whose $(n-j+1)$st bit is 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. The corresponding timing estimate is easily seen to be

$$T_{4,\mathrm{H}} = (\log_2 k)\tau_{\mathrm{H}} + \frac{N}{b_{\mathrm{H}}}. \tag{3.40}$$

(f) *Switch*. Numbering the nodes by binary numbers, we can easily adapt the scatter algorithm devised for the hypercube, on the switch provided the switch can to perform the partial mapping:

$$0^{n-j+1}a_j \rightarrow 0^{n-j}1a_j,$$

where a_j is any $(j-1)$-bit binary number. This is in effect a power of two shift since $0^{n-j}1a_j = 0^{n-j+1}a_j + 2^j$, and is easily realized by most switches. The corresponding time is similar to (3.40):

$$T_{4,\mathrm{S}} = (\log_2 k)\tau_{\mathrm{S}} + \frac{N}{b_{\mathrm{S}}}. \tag{3.41}$$

3.5. Data transposition

This is a scatter operation from each processor. Taking the transpose of a $\sqrt{N} \times \sqrt{N}$ matrix which is originally distributed by rows or columns among the processors is an excellent example of application.

(a) *Broadcast bus*. Each processor broadcasts all its data in turn. Using the result from Section 3.2 we have the time estimate

$$T_{5,\mathrm{B}} = k\left(\tau_{\mathrm{B}} + \frac{N}{kb_{\mathrm{B}}}\right) = k\tau_{\mathrm{B}} + \frac{N}{b_{\mathrm{B}}}.$$

(b) *Shared memory*. All the processors write their data into the shared memory (each processor contributes N/k data) and read the appropriate N/k data back. This requires the time

$$T_{5,\mathrm{M}} = 2\left(\frac{N}{b_{\mathrm{M}}k} + \tau_{\mathrm{M}}\right) max\left(1, \frac{k}{k^*}\right). \tag{3.42}$$

(c) *Ring*. The simplest method is to rotate the data and drop a piece of it after each stop. The corresponding time is

$$T_{5,\mathrm{R}} = \sum_{i=1}^{k} \left(\tau_{\mathrm{R}} + (k-i+1)\frac{N}{b_{\mathrm{R}} k^2} \right) = \frac{1}{2}\frac{N}{b_{\mathrm{R}}} + k\tau_{\mathrm{R}}. \tag{3.43}$$

(d) *Grid*. We can proceed in two stages. First every processor on the main diagonal of the grid of processors, i.e., those with label (i, i), gathers all other blocks of data in the processors belonging to its row. Second each processor on the main diagonal scatters the data to the processors belonging to its columns. This consists of a gather in a ring of $\sqrt{k}$ processors and a scatter in a ring with the same number of processors. From (3.38) the total time is therefore

$$T_{5,\mathrm{G}} = 2\left(\frac{N}{\sqrt{k}\, b_{\mathrm{G}}} + \sqrt{k}\,\tau_{\mathrm{G}} \right). \tag{3.44}$$

(e) *Hypercube*. The idea of the algorithm considered in [13] is as follows. We start by splitting the matrix into 4 equal square blocks. Then the first step of the algorithm consists of exchanging the blocks that are in upper right and lower left positions of the large matrix. In the second step each of the 4 blocks of the resulting matrix is further split into 4 equal parts and the resulting (1, 2) and (2, 1) sub-blocks are again exchanged. The rest of the algorithm can be deduced recursively. The total time required to perform the algorithm is

$$T = \log_2(k)\left[\frac{N}{k b_{\mathrm{H}}} + 2\tau_{\mathrm{H}} \right].$$

However, an optimized algorithm also described in [13] leads to the improved time of

$$T_{5,\mathrm{H}} = \frac{N}{k b_{\mathrm{H}}} + 2\log_2(k)\tau_{\mathrm{H}}.$$

(f) *Switch*. For the switch we can simulate the algorithm employed for the hypercube outlined above. At every step of this algorithm, half of the processors exchange their data packets with the other half. Looking at the detailed description in [13], we observe that we need at every step to connect an arbitrary node X with its neighbor (in the hypercube) $X + 2^i \bmod k$. Here additions are bit-wise. Thus all the nodes whose ith bit is zero must be connected to those whose ith bit is one in a one-to-one fashion. This can be realized by one pass in a shuffle exchange network on which most switches are based. The time that it takes to achieve this algorithm is therefore derived from the above time for the hypercube:

$$T_{5,\mathrm{S}} = \log_2(k)\left[\frac{N}{k b_{\mathrm{S}}} + 2\tau_{\mathrm{S}} \right].$$

4. Summary and conclusion

In Tables 1 and 2 we summarize the timings for the communication algorithms considered in this paper. For convenience we show the upper bounds for the best possible times instead of the actual best times which are not simple to compare. Moreover, for the expressions of the form $(\sqrt{a} + \sqrt{b})^2$ that occur when pipelining is used we will use the inequality $(\sqrt{a} + \sqrt{b})^2 \leqslant 2(a+b)$. Finally, for the shared memory model we assume that $k > k^*$ so that the factor $max(1, k/k^*)$ is always equal to k/k^*.

Although these algorithms are not all optimal, the timings in the tables still give an idea of the performance that can be obtained with each particular architecture. For example, with the

Table 1
Timing estimates for the broadcast bus, the shared memory machine and the ring

	Bus	Shared memory	Ring
One to one	$\frac{N}{b_B} + \tau_B$	$2\frac{N}{b_M} + \tau_M$	$2\frac{N}{b_R} + k\tau_R$
One to all	$\frac{N}{b_B} + \tau_B$	$\left(\frac{N}{b_M} + \tau_M\right)\left(1 + \frac{k}{k^*}\right)$	$2\frac{N}{b_R} + k\tau_R$
All to all	$\frac{N}{b_B} + k\tau_B$	$\frac{N(k+1)}{k^* b_M} + \frac{2}{k^*}\tau_M$	$\frac{N}{b_R} + k\tau_R$
Scatter	$k\tau_B + \frac{N}{b_B}$	$\left(\frac{N}{b_M} + \tau_M\right)\left(1 + \frac{k}{k^*}\right)$	$\frac{N}{2b_R} + \frac{k}{2}\tau_R$
Multiscatter	$\frac{N}{b_B} + k\tau_B$	$2\left(\frac{N}{b_M} + \tau_M\right)\left(1 + \frac{k}{k^*}\right)$	$\frac{N}{2b_R} + k\tau_R$

Table 2
Timing estimates for the grid, the hypercube, and the switch

	Grid	Hypercube	Switch
One to one	$\frac{N}{2b_G} + (\sqrt{k} + 6)\tau_G$	$\frac{2N}{\log k b_H} + (2 + \log_2 k)\tau_H$	$\frac{N}{b_S} + \tau_S$
One to all	$\frac{N}{b_G} + \sqrt{k}\,\tau_G$	$2\left(\frac{N}{b_H} + (\log_2 k)\tau_H\right)$	$4\left(\frac{N}{b_S} + (\log_2 k)\tau_S\right)$
All to all	$\frac{N}{b_G} + 2\sqrt{k}\,\tau_G$	$\frac{2N}{(\log_2 k) b_H} + 2(\log_2 k)\tau_H$	$\frac{N}{b_S} + \log_2(k)\tau_S$
Scatter	$\frac{N}{b_G} + 2\sqrt{k}\,\tau_G$	$\frac{N}{b_H} + (\log_2 k)\tau_H$	$\frac{N}{b_S} + (\log_2 k)\tau_S$
Multiscatter	$2\left(\frac{N}{b_G} + \sqrt{k}\,\tau_G\right)$	$\frac{N}{k b_H} + 2\log_2(k)\tau_H$	$\log_2(k)\left[\frac{N}{k b_S} + 2\tau_S\right]$

assumptions made, when large amounts of data are involved, i.e., when the contribution of startup times is negligible, hypercubes seem to perform best on every operation except the broadcast operation which is clearly best performed by a broadcast bus architecture. At the other extreme, when the amount of data N is very small and startup times dominate, the table reveals that roughly speaking the ring and the shared memory architectures have the largest times $O(k)$ followed by the grid architecture ($O(\sqrt{k})$ and the hypercube, the switch ($O(\log_2 k)$). The performance of the bus model is mixed in this case.

References

[1] K.E. Batcher, The FLIP network in STARAN, in: *Proc. 1976 IEEE International Conference on Parallel Processing* (1976) 65–71.

[2] W. Chuan-Lin, On a class of multistage interconnection networks, *IEEE Trans. Comput.* **29** (1980) 694–702.

[3] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Williken and T. Blackdar, Performance measurements on a 128-node butterfly parallel processor, in: D. Degroot, ed., *Proc. 1985 IEEE/ACM International Conference on Parallel Processing* (1985) 531–540.

[4] J.M. Frailong, W. Jalby and J. Lenfant, XOR schemes: A flexible data organization in parallel memories, in: *Proc. 1985 IEEE International Conference on Parallel Processing* (1985) 276–283.

[5] D. Gannon and J. Van Rosendale, On the impact of communication complexity in the design of parallel algorithms, Technical Report 84–41, ICASE, 1984.

[6] W.M. Gentleman, Some complexity results for matrix computation on parallel processors, *J. ACM* **25** (1978) 112–115.
[7] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer—Designing an MIMD shared memory parallel computer, *IEEE Trans. Comput.* **32** (1983) 175–89.
[8] I.C.F. Ipsen, Y. Saad and M.H. Schultz, Complexity of dense linear system solution on a multiprocessor ring, *Lin. Alg. Appl.* **77** (1986) 205–239.
[9] D.H. Lawrie, Access and alignment of data in an array processor, *IEEE Trans. Comput.* **24** (1975) 1145–55.
[10] Y. Saad, Communication complexity of the Gaussian elimination algorithm on multiprocessors, *Lin. Alg. Appl.* **77** (1986) 315–340.
[11] Y. Saad and M.H. Schultz, Topological properties of hypercubes, *IEEE Trans. Comput.* **37** (1988) 867–872.
[12] Y. Saad and M.H. Schultz, Direct parallel methods for solving banded linear systems, *Lin. Alg. Appl.* **88** (1987) 623–650.
[13] Y. Saad and M.H. Schultz, Data communication in hypercubes, *J. Parallel Distrub. Comput.* **6** (1989) 115–135.
[14] A.H. Sameh, Solving the linear least squares problem on a linear array of processors, in: *Proc. Purdue Workshop on Algorithmically-Specialized Computer Organizations*, W. Lafayette, IN (Academic Press, New York, 1982).
[15] H.D. Shapiro, Theoretical limitations on the efficient use of parallel memories, *IEEE Trans. Comput.* **27** (1978) 421–28.
[16] H.J. Siegel, *Interconnection Networks for Large Scale Parallel Processing* (Lexington Books, Lexington, MA, 1985).