Yale University
Department of Computer Science

Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures

S. Lennart Johnsson and Ching-Tien Ho

YALEU/DCS/TR-572
September 1987

This work has been supported in part by the Office of Naval Research under Contracts N00014-84-K-0043 and N00014-86-K-0564. Approved for public release: distribution is unlimited.

† A revised edition of TR-494. To appear in *SIAM Journal on Algebraic and Discrete Methods*.

Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures

S. Lennart Johnsson and Ching-Tien Ho
Department of Computer Science
Yale University
New Haven, CT 06520

Abstract. In a multiprocessor with distributed storage the data structures have a significant impact on the communication complexity. In this paper we present a few algorithms for performing matrix transposition on a Boolean n-cube. One algorithm performs the transpose in a time proportional to the lower bound both with respect to communication start-ups and element transfer times. We present algorithms for transposing a matrix embedded in the cube by a binary encoding, a *binary-reflected* Gray code encoding of rows and columns, or combinations of these two encodings. The transposition of a matrix when several matrix elements are identified to a node by *consecutive* or *cyclic* partitioning is also considered and lower bound algorithms given. Experimental data are provided for the Intel iPSC and the Connection Machine.

Key words. matrix transpose, Boolean cubes, personalized communication, routing, data encoding

AMS(MOS) subject classification. 65F30, 68P99, 68Q20, 68Q25

1 Introduction

Matrix transposition is a permutation frequently performed in linear algebra. It is useful in the solution of systems of linear equations by a variety of techniques. For instance, the solution of partial differential equations by the Alternating Direction Method (ADM) is typically carried out by transposing the data between the solution phases in the different directions. Such data transposition may also be beneficial with respect to performance for the ADM on Boolean n-cube configured architectures, even though multi-dimensional arrays can be embedded in Boolean cubes preserving proximity [13, 14]. Another example where data transposition may be advantageous is in the solution of Poisson's problem by the Fourier Analysis Cyclic Reduction (FACR) method. Matrix transposition can also be used to realize arbitrary permutations [21,20].

In this paper we focus on matrix transposition on Boolean n-cube architectures. The transpose can be formed recursively as described in [19,1,9,15]. Stone describes a mapping to shuffle-exchange networks for the case with one matrix element per node. We consider the case with multiple matrix elements per node and focus on the pipelining of communication operations and the optimal use of the communication bandwidth of the Boolean n-cube. In [9,10] we described and analyzed the complexity of a transpose algorithm for a two-dimensional mesh and presented a few algorithms for the transposition of matrices embedded in the cube by binary or Gray code encoding of the row and column indices. In this paper we present a transpose algorithm that is of lower complexity in the case of concurrent communication on multiple ports, and present experimental data for the Intel iPSC and the Connection Machine [3].

We first introduce the notation and data structures used in this study, then present algorithms for the transpose operation for one-dimensional and two-dimensional partitionings. Implementation issues particular to the actual machines used, but important for the interpretation of the experimental results presented, are addressed after the description of the algorithms. A summary and conclusion follows.

2 Preliminaries

Let A be a $P \times Q$ matrix. Throughout the paper, we assume that $P = 2^p$ and $Q = 2^q$. The number of bits required for the encoding of the matrix elements is $m = p + q$. The transpose A^T of A is defined by the relation $a^T(u,v) = a(v,u)$, where $a^T(u,v)$ is the element in row u and column v of A^T, and $a(u,v)$ is the element of A in row u and column v. Let the binary encoding of u be $(u_{p-1}u_{p-2}\ldots u_0)$ and the binary encoding of v be $(v_{q-1}v_{q-2}\ldots v_0)$. Then the address of element $a(u,v)$ is naturally defined to be $(u_{p-1}u_{p-2}\ldots u_0 v_{q-1}v_{q-2}\ldots v_0) = (w_{m-1}w_{m-2}\ldots w_0)$, or $(u||v) = w$ for short, where '$||$' is the concatenation operator for binary numbers.

Definition 1 *The matrix transposition operation is the permutation* $loc(u_{p-1}u_{p-2}\ldots u_0\, v_{q-1}v_{q-2}\ldots v_0) \leftarrow loc(v_{q-1}v_{q-2}\ldots v_0 u_{p-1}u_{p-2}\ldots u_0)$ *where* $loc(w)$ *is the memory location of element* w.

Note that we arbitrarily assumed that the p highest order dimensions are used for the encoding of row indices. We use this assumption throughout this paper, but any other subset of p dimensions could have been used.

With the assumption above the p highest order dimensions encode row indices before the transposition and the q highest order dimensions encode column indices after the transposition. A vector transposition requires no data movement. For the matrix transposition it is sometimes appropriate to consider a square array of $2\max(p,q)$ dimensions.

Definition 2 *A $P \times Q$ matrix with $P > Q$ is extended to a square matrix by introducing* virtual elements *corresponding to $P - Q$ columns. The extension is made similarly if $P < Q$.*

The extension can be made by adding columns corresponding to high or low order dimensions of the column address space, or by mixing columns of virtual elements with columns of real elements. Whichever alternative is preferable depends on the particular transposition algorithm, and data assignment scheme (described later).

Definition 3 *A* shuffle *operation, sh^1 on a set of elements $\mathcal{W}$ with addresses w, $w \in \{0, 1, \ldots, 2^m - 1\}$ encoded in binary representation $(w_{m-1}w_{m-2}\ldots w_0)$ is a permutation defined by a one step* left cyclic shift, *$loc(w_{m-1}w_{m-2}\ldots w_0) \leftarrow loc(w_{m-2}w_{m-3}\ldots w_0 w_{m-1})$, $w \in \{0, 1, \ldots, 2^m - 1\}$. An* unshuffle *operation, sh^{-1} is defined by a one step right cyclic shift. $sh^k = sh\ sh^{k-1}$ is a k step left cyclic shift.*

Clearly, $sh^1 sh^{-1} = I$, where I is the identity operator. Also, $sh^k(w) = sh^{-(m-k)}(w)$.

Lemma 1 *Let A be a $2^p \times 2^q$ matrix. $A^T \leftarrow sh^p A$, or $A^T \leftarrow sh^{-q} A$.*

Corollary 1 *On a shuffle-exchange network of $N = 2^n$ nodes, $n = p + q$, and bi-directional communication links, the matrix transposition requires at most* $\min(p, q)$ *communication steps.*

A shuffle-exchange network has all the connections corresponding to the sh^1 operation, and connections from every even node to the succeeding odd node.

Definition 4 *Let $w = (w_{m-1}w_{m-2}\ldots w_0)$ and $z = (z_{m-1}z_{m-2}\ldots z_0)$. Then $Hamming(w, z) = \sum_{i=0}^{m-1}(w_i \oplus z_i)$, where $\oplus$ is the exclusive or operation.*

Lemma 2 *For m even there exist at least one w such that $Hamming(w, sh^1 w) = m$, and for m odd $Hamming(w, sh^1 w) = m - 1$. In general, for k shuffles*

$$\max_w Hamming(w, sh^k w) = \begin{cases} m, & \frac{m}{\gcd(m,k)} \text{ is even;} \\ m - \gcd(m, k), & \frac{m}{\gcd(m,k)} \text{ is odd.} \end{cases}$$

Proof: For m even, let $w = (0101\ldots01)$. Then $Hamming(w, sh^1 w) = m$. For m odd, let $w = (0101\ldots010)$. Then $Hamming(w, sh^1 w) = m - 1$. Note that w and $sh^1 w$ contain the same numbers of 0's and 1's. Since one of them is odd, the Hamming distance between w and $sh^1 w$ is at most $m - 1$. For k shuffles, the bits can be divided into $\gcd(m, k)$ groups of bit strings of length $\frac{m}{\gcd(m,k)}$. The lemma follows. ∎

Corollary 2 *For m even* $\max_w$ *Hamming*$(w, sh^{\frac{m}{2}} w) = m$.

Lemma 3 *For* $0 \leq k < m$, $\max_w$ *Hamming*$(w, sh^k w) \geq k$.

Proof: Since $m > k$ we have $\frac{m}{\gcd(m,k)} > \frac{k}{\gcd(m,k)}$ or $\frac{m}{\gcd(m,k)} \geq 1 + \frac{k}{\gcd(m,k)}$. This means $m - \gcd(m,k) \geq k$. Lemma 2 completes the proof. ▮

Definition 5 *Let* $x = (x_{n-1}x_{n-2}\ldots x_0)$, $x_i \in \{0,1\}$, $\forall i \in \{0,1,\ldots,n-1\}$ *be the address of a node in a Boolean n-cube. Then node x is connected to nodes in the set* $\{(x_{n-1}x_{n-2}\ldots\overline{x_i}\ldots x_0)|\forall i \in \{0,1,\ldots,n-1\}\}$.

A Boolean n-cube has $N = 2^n$ nodes, and each node n neighbors. The diameter is n and the number of links is $\frac{1}{2}nN$. There exist n paths between any pair of nodes (x,y). Of these paths *Hamming*(x,y) paths are of length *Hamming*(x,y) and $n -$ *Hamming*(x,y) paths are of length *Hamming*$(x,y) + 2$ [18]. We will use this property in devising transposition algorithms with multiple paths between source and destination processors for minimization of the data transfer time.

Lemma 4 *Matrix transposition on a Boolean n-cube requires at least as many communication steps as the transposition on a shuffle-exchange network.*

Lemma 4 is immediate from lemma 3.

In general, the number of matrix elements may be larger than the number of processors, and several matrix elements must be allocated to the storage of individual processors. We assume that the matrix elements are distributed evenly among the processors. For $n \leq \max(p,q)$ there is a choice between one- and two-dimensional partitioning. For either kind of partitioning the matrix elements can be assigned to processors *cyclicly*, or *consecutively* [9,10], or by a *combined* assignment scheme.

Definition 6 *In a one-dimensional cyclic partitioning on N processors, row u (column v) is assigned to processor* $u \bmod N$ *(*$v \bmod N$*) and in a one-dimensional consecutive partitioning row u (column v) is assigned to processor* $\lfloor \frac{u}{\lceil \frac{P}{N} \rceil} \rfloor$ *(*$\lfloor \frac{v}{\lceil \frac{Q}{N} \rceil} \rfloor$*).*

Corollary 3 *In an n-cube the n lowest order bits of the binary encoded row (column) index determines the processor to which a row (column) is assigned in the cyclic partitioning. In the consecutive assignment the n highest order bits determines the processor assignment, if the number of rows (columns) is a power of two.*

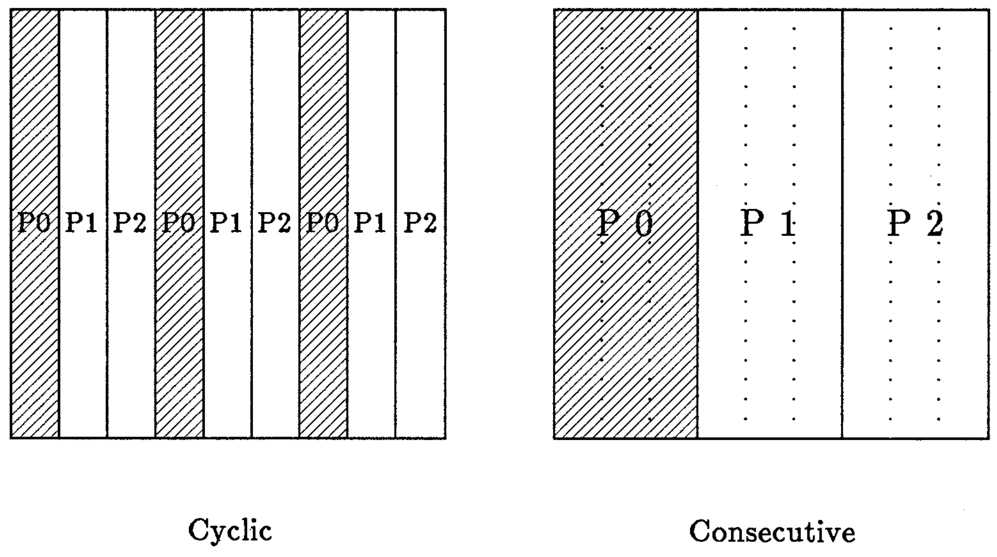


Figure 1: Cyclic and Consecutive one-dimensional partitioning.

The dimensions that are of higher (lower) order than the real processor address field are used for cyclic (consecutive) assignment. The notions of cyclic and consecutive assignment are relative to a given real processor address field.

In the two-dimensional partitioning we let $N_r = 2^{n_r} \leq P$ denote the number of partitions in the row direction and $N_c = 2^{n_c} \leq Q$ the number of partitions in the column direction. The total number of partitions is $N_r \times N_c \leq N$ $(n_r + n_c \leq n)$. In the cyclic partitioning matrix element (u, v) is assigned to partition $(u \bmod N_r, v \bmod N_c)$ and in the consecutive partitioning it is assigned to partition $(\lfloor \frac{u}{\lceil \frac{P}{N_r} \rceil} \rfloor, \lfloor \frac{v}{\lceil \frac{Q}{N_c} \rceil} \rfloor)$, Figure 2. For a matrix partitioned by cyclic assignment the n_r lowest order bits of the matrix row index determines the processor row index. Analogously, the n_c lowest order bits of the matrix column index determines the processor column index. In consecutive storage, the n_r highest order bits in the matrix row index determines the processor row index and the n_c highest order bits of the column index determines the processor column index, since P and Q are powers of two.

The cyclic and consecutive assignment schemes are illustrated in Figures 1 and 2 with respect to the matrix elements.

Definition 7 *The part of the address field that does not correspond to real processors defines* virtual processors.

The cyclic and consecutive assignment schemes with respect to the address space is as follows:

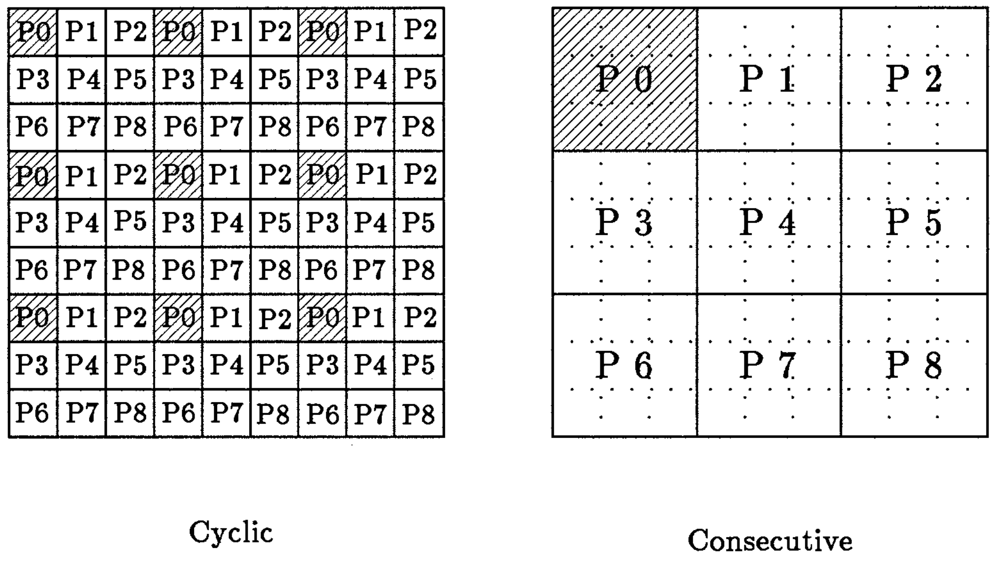

P0 P1 P2 P0 P1 P2 P0 P1 P2
P3 P4 P5 P3 P4 P5 P3 P4 P5
P6 P7 P8 P6 P7 P8 P6 P7 P8
P0 P1 P2 P0 P1 P2 P0 P1 P2
P3 P4 P5 P3 P4 P5 P3 P4 P5
P6 P7 P8 P6 P7 P8 P6 P7 P8
P0 P1 P2 P0 P1 P2 P0 P1 P2
P3 P4 P5 P3 P4 P5 P3 P4 P5
P6 P7 P8 P6 P7 P8 P6 P7 P8
P 0 P 1 P 2
P 3 P 4 P 5
P 6 P 7 P 8
Cyclic
Consecutive

Figure 2: Cyclic and Consecutive two-dimensional partitioning.

One-dimensional cyclic column partitioning

$$\big(\underbrace{u_{p-1}u_{p-2}\dots u_0\ v_{q-1}v_{q-2}\dots v_{n_c}}_{vp}\underbrace{v_{n_c-1}\dots v_0}_{rp}\big).$$

One-dimensional consecutive column partitioning

$$\big(\underbrace{u_{p-1}u_{p-2}\dots u_0}_{vp}\ \underbrace{v_{q-1}v_{q-2}\dots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}\dots v_0}_{vp}\big).$$

For the cyclic two-dimensional assignment the address field is partitioned as

$$\big(\underbrace{u_{p-1}u_{p-2}\dots u_{n_r}}_{vp}\underbrace{u_{n_r-1}\dots u_0}_{rp}\ \underbrace{v_{q-1}v_{q-2}\dots v_{n_c}}_{vp}\underbrace{v_{n_c-1}\dots v_0}_{rp}\big),$$

and for the consecutive assignment the address field is partitioned as

$$\big(\underbrace{u_{p-1}u_{p-2}\dots u_{p-n_r}}_{rp}\underbrace{u_{p-n_r-1}\dots u_0}_{vp}\ \underbrace{v_{q-1}v_{q-2}\dots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}\dots v_0}_{vp}\big),$$

where *vp* denotes the dimensions of the address space used for virtual processor addresses and *rp* denotes the dimensions used for real processor addresses. The number of dimensions used for the consecutive, or cyclic mapping is $m - n_c$ (or $m - n_r$) in the one-dimensional case and $m - n_r - n_c$ in the two-dimensional case. For column partitioning, $n_r = 0$, $0 \leq n_c \leq n$. For row partitioning, $n_c = 0$, $0 \leq n_r \leq n$.

The *cyclic* and *consecutive* storage forms are two extreme cases of real processor assignment. We refer to the general case as *combined* assignment. Any subset of dimensions of the address space can be used for real processor addresses. As an example of combined assignment we consider the storage of a banded matrix for the equation solvers in [8,12]. The non-zero elements of the matrix, the right hand sides, and the solution vectors can be stored in a rectangular array by conventional row/column storage of the matrix, or by row/diagonal organization. We do not here discuss the techniques for band matrix storage, and their consequences for the solution procedure. For the illustration here we simply assume that the relevant elements are stored in an array of $P = 2^p$ rows and $Q = 2^q$ columns. Then, for a two-dimensional partitioning with 2^{n_c} processors in both the row and column directions, blocks of size $2^{q-n_c} \times 2^{q-n_c}$ elements may be stored in the same processor, and blocks assigned cyclically with respect to the row addresses i.e., the address field is partitioned as

$$(\underbrace{u_{p-1}u_{p-2}\dots u_q}_{vp}\underbrace{u_{q-1}\dots u_{q-n_c}}_{rp}\underbrace{u_{q-n_c-1}\dots u_0}_{vp}\ \underbrace{v_{q-1}\dots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}\dots v_0}_{vp}).$$

The total number of real processor dimensions is $2n_c$. For the row assignment the n_c contiguous dimensions of the address field used for real processor addresses divides the address space into two parts: one part of $q - n_c$ dimensions used for consecutive assignment and $p - q$ dimensions used for cyclic assignment. For the concurrent elimination of multiple vertices the matrix is partitioned into S block rows. With $S = 2^s$ the s highest order bits of the matrix row addresses are used for real processor addresses. With the previous assignment for each such block the address field is partitioned as

$$(\underbrace{u_{p-1}u_{p-2}\dots u_{p-s}}_{rp}\underbrace{u_{p-s-1}u_{p-2}\dots u_q}_{vp}\ \underbrace{u_{q-1}\dots u_{q-n_c}}_{rp}\underbrace{u_{q-n_c-1}\dots u_0}_{vp}\ \underbrace{v_{q-1}\dots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}\dots v_0}_{vp}).$$

Hence, in this case the dimensions used for real processor addresses forms two fields. The number of dimensions for real processors in the row direction is $s + n_c$, and the total number of real processor dimensions $s+2n_c$. The notions of cyclic and consecutive partitioning are now conditioned on the part of the real processor address fields.

We now turn to the communication required for matrix transposition. Consider a one-dimensional partitioning such that $p = q > n_c = n$ and cyclic partitioning by columns before the transposition. Then every processor sends 2^{m-2n} elements to every other processor. *All-to-all personalized communication* [5,7] is required. To see this fact note that there are 2^{m-n} virtual processors per real processor, and that the address field prior to the transposition is partitioned as

$$(\underbrace{u_{p-1}u_{p-2}\dots u_0\ v_{q-1}v_{q-2}\dots v_n}_{vp}\underbrace{v_{n-1}\dots v_0}_{rp}).$$

After the transposition the partitioning is

$$(\underbrace{v_{q-1}v_{q-2}\dots v_0\ u_{p-1}u_{p-2}\dots u_n}_{vp}\underbrace{u_{n-1}\dots u_0}_{rp}),$$

which in the original address field is

$$(\underbrace{u_{p-1}u_{p-2}\ldots u_n}_{vp}\underbrace{u_{n-1}\ldots u_0}_{rp}\ \underbrace{v_{q-1}v_{q-2}\ldots v_0}_{vp}).$$

Hence, the row address field in the initial allocation is partitioned into 2^n partitions for each column, and each such partition sent to a unique processor for the matrix transposition. The address fields for real processors before the transposition and after the transposition are disjoint.

If $q < n \leq p$ and the initial assignment is by columns, then only 2^q processors are used before the transposition, but all 2^n processors used after the transposition. The number of virtual processors per real processor before the transposition is 2^p, and after the transposition 2^{m-n}. The row address field is divided into 2^n partitions. The address fields for real processors before and after the transposition are disjoint. The transposition is accomplished by all 2^q processors holding matrix elements sending a unique set of data to each of the 2^n processors. The communication is *some-to-all personalized communication.* The reverse operation is *all-to-some personalized communication.* In the extreme case such as transposing a vector, it is *one-to-all* or *all-to-one personalized communication.* In a two-dimensional partitioning with the same number of processors assigned to rows and columns, and the same assignment scheme (cyclic or consecutive) for rows and columns, the address fields for real processors before and after the transposition are the same. The communication is between distinct pairs of processors.

One of the reasons for not using all processors before or after a transposition is that the number of dimensions for the row or column address field is smaller than the number of processors dimensions assigned to that address field. *Virtual elements* can be introduced to simplify the analysis. *Virtual processors* define local storage addresses.

Let $\mathcal{R}$ be the set of dimensions used for real processors, and $\mathcal{V}$ the set of dimensions used for virtual processors: $\mathcal{R} = \{d_i | i = 0, 1, \ldots, rp-1\}$ and $\mathcal{V} = \{d'_i | i = 0, 1, \ldots, vp-1\}$, where $d_i, d'_i \in \{0, 1, \ldots, m\}$. Furthermore $\mathcal{R} \cap \mathcal{V} = \phi$ and $\mathcal{R} \cup \mathcal{V} = \{0, 1, \ldots m-1\}$. The number of dimensions used for real processor addresses is $|\mathcal{R}| = rp$, and the number of dimensions used for virtual processors is $|\mathcal{V}| = vp$ $(rp + vp = m)$. Denote the set of dimensions of the matrix encoding assigned to real processors before the transposition by $\mathcal{R}_b$ and the set of matrix dimensions used for real processors after the transposition by $\mathcal{R}_a$. Let $\mathcal{I} = \mathcal{R}_b \cap \mathcal{R}_a$.

Clearly, for any one-dimensional partitioning $\mathcal{I} = \phi$.

We have so far assumed that the matrix elements are embedded in the set of processors by a binary encoding. Such an embedding does not preserve proximity. A *binary-reflected Gray code* [16] encoding of row and column indices preserves adjacency. This code is referred to as Gray code in the following and the encoding of w is $G(w)$. The conversion from one kind of encoding to the other can be accomplished in $n-1$ routing steps with additional local data rearrangement. The paths in the routing can

Enc./Part.	Consecutive	Cyclic
Binary, Row	$(u_{p-1}u_{p-2}...u_{p-n})$	$(u_{n-1}u_{n-2}...u_0)$
Binary, Column	$(v_{q-1}v_{q-2}...v_{q-n})$	$(v_{n-1}v_{n-2}...v_0)$
Gray, Row	$(G(u_{p-1}u_{p-2}...u_{p-n}))$	$(G(u_{n-1}u_{n-2}...u_0))$
Gray, Column	$(G(v_{q-1}v_{q-2}...v_{q-n}))$	$(G(v_{n-1}v_{n-2}...v_0))$

Table 1: The processor address for matrix element $(u_{p-1}u_{p-2}\dots u_0, v_{q-1}v_{q-2}\dots v_0)$ with *consecutive* and *cyclic* encodings.

Enc./Part.	Combined	
	Contiguous	Non-contiguous
Binary, Row	$(u_{p-i}u_{p-i-1}...u_{p-i-n+1})$	$(u_{p-1}...u_{p-s}u_{n-s-1}...u_0)$
Binary, Column	$(v_{q-i}v_{q-i-1}...v_{q-i-n+1})$	$(v_{q-1}...v_{q-s}v_{n-s-1}...v_0)$
Gray, Row	$(G(u_{p-i}u_{p-i-1}...u_{p-i-n+1}))$	$(G(u_{p-1}...u_{p-s})G(u_{n-s-1}...u_0))$
Gray, Column	$(G(v_{q-i}v_{q-i-1}...v_{q-i-n+1}))$	$(G(v_{q-1}...v_{q-s})G(v_{n-s-1}...v_0))$

Table 2: The processor address for matrix element $(u_{p-1}u_{p-2}\dots u_0, v_{q-1}v_{q-2}\dots v_0)$ with two examples of *combined* encoding.

be made to be edge-disjoint [9].

Adjacency is of no concern for virtual processor addresses in a storage with uniform access time, but may be of significance for interprocessor communication, in particular for Boolean cube configured multiprocessors. It is possible to restrict the Gray code encoding to the real processor address field. For instance, in the consecutive assignment the stripes/blocks can be assigned to processors by a Gray code encoding, while the elements within the stripes/blocks are ordered in the binary order.

Considering binary and Gray code encoding of the processor address field, and consecutive, cyclic, or combined assignment with a consecutive or split address field a total of 16 matrix embeddings result for a one-dimensional partitioning. The conversions between any two of the 16 assignment schemes are equivalent, i.e., *all-to-all personalized communication*, in terms of the global communication, if $\mathcal{I} = \phi$ and $|\mathcal{R}_a| = |\mathcal{R}_b| = |\mathcal{R}|$. Table 1 shows the real address fields and their encoding in terms of the matrix dimensions for *consecutive* and *cyclic* assignments. Table 2 shows the encodings for two examples of *combined* assignment. The general case for which n arbitrarily chosen dimensions are used for real processor addresses is treated in [4].

For the architecture we assume that the communication is packet oriented with a communications overhead τ, a transmission time per element t_c, and a maximum packet size of B_m elements. A communications overhead is incurred for each communications link traversed. For a bit-serial architecture, such as the Connection Machine, the overhead is only incurred once through pipelining. With the operating system for the Intel iPSC on which our experiments were carried out $\tau \approx 5$ $m$sec, $t_c \approx 1$ μsec/byte and $B_m = 1$ $k$bytes. For the algorithm description and analysis we consider two cases with respect to communication capabilities: communication restricted to one port at a time, *one-port* communication, and concurrent communication on all ports, *n-port* communication. *One-port* communication is a good approximation of the capabilities of the Intel iPSC. Furthermore, we assume bi-directional communication, i.e., that a processor can send and receive data concurrently on the same port. Therefore, one send *or* one receive operation takes the same time as one exchange operation of two adjacent nodes through the same link for both *one-port* and *n-port* communications.

3 Generic Algorithms

3.1 One-to-All Personalized Communication

In [7] we devised and analyzed algorithms for *one-to-all* and *all-to-all personalized communication. One-to-all personalized communication* can be performed in a time within a factor of two of the lower bound by routing according to a *Spanning Binomial Tree* (SBT) with *one-port* communication [17,2,5]. Before the communication the source node holds all PQ data elements. After the communication, every processor holds $\frac{PQ}{N}$ data elements. The communication time for SBT routing and scheduling all data for a subtree at once [5] is $T = (1 - \frac{1}{N})PQt_c + \sum_{i=1}^{n}\lceil\frac{PQ}{2^i B_m}\rceil\tau$, which is minimized for $B_m \geq \frac{PQ}{2}$. $T_{min} = (1 - \frac{1}{N})PQt_c + n\tau$. The lower bound $T_{l\ b} \geq \max((1 - \frac{1}{N})PQt_c, n\tau) \geq \frac{1}{2}((1 - \frac{1}{N})PQt_c + n\tau)$.

With *n-port* communication routing according to a SBT results in a time complexity of an order higher than the lower bound. Half of the nodes of a SBT are in one of the subtrees of the root node, and the minimum transmission time is $\frac{1}{2}PQt_c$. The lower bound for *n-port* communication is $T_{l\ b} \geq \max(\frac{1}{n}(1-\frac{1}{N})PQt_c, n\tau) \geq \frac{1}{2}(\frac{1}{n}(1-\frac{1}{N})PQt_c+n\tau)$. One routing strategy optimal within a small constant factor is to use a *Spanning Balanced n-Tree* (SBnT) [5,7,6]. The communication time for SBnT routing and scheduling data for each subtree in a reverse breadth-first order is $T \approx \sum_{i=1}^{n}(\frac{1}{n}\binom{n}{i}\frac{PQ}{N}t_c+\lceil\frac{1}{n}\binom{n}{i}\frac{PQ}{B_m N}\rceil\tau) = T \approx \frac{1}{n}(1-\frac{1}{N})PQt_c+\sum_{i=1}^{n}(\lceil\frac{1}{n}\binom{n}{i}\frac{PQ}{B_m N}\rceil\tau)$, which has a minimum of $T_{min} = \frac{1}{n}(1-\frac{1}{N})PQt_c+n\tau$ for $B_m \geq \max_{\forall i}\binom{n}{i}\frac{1}{n}\frac{PQ}{N} \approx \sqrt{\frac{2}{\pi}}\frac{PQ}{n^{3/2}}$. The speedup of the transmission time of the SBnT routing over the SBT routing is a factor of $\frac{1}{2}n$. The maximum packet size is reduced approximately by a factor of n.

In the SBnT routing the node set is divided into n approximately equal sets. An alternative routing for *n-port* communication, is to divide the data set $(\frac{PQ}{N})$ for each node into n equal parts and route the parts according to SBT's *rotated* with respect to each other, if $\frac{PQ}{N} \bmod n = 0$.

Definition 8 *A graph is* rotated *with respect to another graph, if all its addresses are obtained through the same number of shuffle operations, sh^k for some k, of the addresses of the other graph.*

Definition 9 *A graph is a* reflection *of another graph, if all its addresses are obtained through a bit-reversal of the addresses of the other graph.*

Note that in the case of the SBT a reflected SBT can be obtained by complementing trailing zeroes, instead of leading zeroes. The minimum time for *one-to-all personalized communication* using n distinctly rotated spanning binomial trees and scheduling data for each subtree in a reverse breadth-first order is $T_{min} = \frac{1}{n}(1 - \frac{1}{N})PQt_c + n\tau$ [5]. This complexity is of the same order as that of the lower bound. The minimum time is achieved for $B_m \geq \sqrt{\frac{2}{\pi}\frac{PQ}{n^{3/2}}}$. A similar algorithm of the same complexity was also derived independently by Stout et al. [21,20].

For $\frac{PQ}{N} = k < n$ the SBnT routing has a lower time complexity for element transfers. For k SBT's the transfer time for optimally rotated spanning binomial trees is $(2^n - 1)\frac{2^{\frac{n}{k}-1}}{2^{\frac{n}{k}}-1}\frac{PQ}{N}t_c$ and for optimally reflected and rotated spanning binomial trees the minimum transfer time with concurrent communication on all ports is $(2^n - 1)\frac{2^{\frac{2n}{k}-1}+1}{2^{\frac{2n}{k}}-1}\frac{PQ}{N}t_c$. For $k = 2$ reflection yields a maximum of $\frac{1}{2}N + 1$ element transfers over any edge (and a minimum of $\sqrt{2N}$). Rotation yields a maximum of $\frac{1}{2}N + \sqrt{\frac{1}{2}N}$ element transfers over any edge. For $k = 2$ the optimum rotation is by $\frac{1}{2}n$ steps. In general, the optimum rotation is by $\frac{n}{k}$ steps for $\frac{PQ}{N} = k < n$, if n is a multiple of k.

3.2 All-to-All Personalized Communication

For *all-to-all personalized communication* a simple exchange algorithm scanning through the dimensions of the cube for *one-port* communication requires a time $T = n\frac{PQ}{2N}t_c + n\lceil\frac{PQ}{B_m 2N}\rceil\tau$, which has the minimum $T_{min} = n(\frac{PQ}{2N}t_c + \tau)$ for $B_m \geq \frac{PQ}{2N}$, [17,9,15,7,2]. In each communication $\frac{PQ}{2N}$ elements are exchanged. The exchange algorithm routes elements from a node to all other nodes according to a SBT. The SBT's rooted at different nodes are *translations* of each other. A tree rooted at node s is a translation of the tree rooted at node 0, if the addresses of the nodes in the tree rooted at node s are obtained through a bit-wise exculsive-or operation, $x \oplus s$, for every node x of the tree rooted at node 0. In the exchange algorithm the dimensions of the cube can be scanned in an arbitrary order. Starting with the highest order dimension of the real processor address and virtual processor address before the communication, a single block is communicated in the first transfer. The number of blocks doubles for each step of the exchange algorithm, and the block size is reduced by a factor of 2.

This exchange algorithm can be explained in terms of the address space of the data set subject to *all-to-all personalized communication.* Let the data assignment before and after the communication be

$$Before: \quad (\underbrace{w_{m-1}w_{m-2}\ldots w_{rp}}_{vp}\,\underbrace{w_{rp-1}\ldots w_0}_{rp}),$$

$$After: \quad (\underbrace{w_{m-1}w_{m-2}\ldots w_s}_{vp}\,\underbrace{w_{s-1}w_{s-2}\ldots w_{s-rp}}_{rp}\,\underbrace{w_{s-rp-1}\ldots w_{rp}w_{rp-1}\ldots w_0}_{vp}).$$

Then, in the i^{th} exchange step real processor dimension $rp-i-1$ and virtual processor dimension $s-i-1$, $i \in \{0,1,\ldots,rp-1\}$ are involved in the exchange.

Exchange step i:

$$(\underbrace{w_{m-1}w_{m-2}\ldots w_s}_{vp}\,\underbrace{w_{s-1}\ldots w_{s-i+1}}_{rp}\,w_{s-i}\,\underbrace{w_{s-i-1}\ldots w_{rp-i+1}}_{vp}\,w_{rp-i}\,\underbrace{w_{rp-i-1}\ldots w_0}_{rp}).$$

The data volume in each exchange remains constant, since the number of virtual processor dimensions remain constant. But, the exchange dimension partitions the virtual address space into an increasing number of smaller blocks for increasing i. A shuffle operation on the virtual addresses between each exchange operation would allow the exchange operation to always work with single block exchanges. The shuffle operation implies extensive local data movement.

As an alternative to a local shuffle operation in order to minimize the number of communication start-ups blocks can be moved to a buffer, and a number of blocks sent in the same communication. For the Intel iPSC moving data to a buffer requires a significant time, and there exists a block size less than the buffer size for which the copy time is greater than the start-up time. We devised an optimal buffer scheme that is presented in connection with the discussion of our experiments on the Intel iPSC.

Definition 10 *The "Standard Exchange Algorithm" on $2l$ dimensions performs an exchange of data between dimensions $g(i)$ and $f(i)$, where the sequences $\{g(i)\}$ and $\{f(i)\}$, $i \in \{0,1,\ldots,l-1\}$, are disjoint, and both monotonically increasing, or decreasing, as a function of i. The exchange is made on data such that $w_{g(i)} \oplus w_{f(i)} = 1$.*

For instance, $g(i) = s-i-1$ and $f(i) = rp-i-1$ for the above example. If $p = q$, $g(i) = m-1-i$, $f(i) = q-1-i$, and $2l = m$, then the standard exchange algorithm realizes a matrix transposition. There is no particular need to restrict the exchanges to proceed from higher to lower order dimensions, or lower to higher order dimensions on both virtual and real processor dimensions. By allowing exchanges on arbitrarily paired real and virtual processor dimensions various forms of data conversions can be accomplished. We will give a few examples later. For a general discussion see [4].

Definition 11 *The "General Exchange Algorithm" on $2l$ dimensions performs an exchange between dimensions $g(i)$ and $f(i)$, where $(g(i), f(i))$ is an arbitrary pair of dimensions such that $g(i) \neq g(j)$, $f(i) \neq f(j)$, $i \neq j$, $\forall i, j \in \{0, 1, \ldots, l-1\}$. An exchange is made on data such that $w_{g(i)} \oplus w_{f(i)} = 1$.*

Note that the sets $\{g(i)\}$ and $\{f(i)\}$ are not necessarily disjoint and the sequences $g(0), g(1), \ldots, g(l-1)$ and $f(0), f(1), \ldots, f(l-1)$ are not necessarily increasing or decreasing. The *general exchange algorithm* can be applied to the *bit-reversal* permutation as described in section 7 and the k shuffle operation described in [4].

With *n-port* communication pipelining can be employed in the exchange algorithm, but the algorithm so modified is suboptimal. However, routing based on spanning balanced n-trees, or rotated spanning binomial trees, and scheduling of data for subtrees in either postorder, or reverse breadth-first order, only requires a time of $T_{min} = \frac{PQ}{2N}t_c + n\tau$ [7]. A similar algorithm of the same complexity was also derived independently by Stout et al. [21,20]. This complexity is again within a factor of 2 of the lower bound $T_{l\ b} \geq \max(\frac{PQ}{2N}t_c, n\tau) \geq \frac{1}{2}(\frac{PQ}{2N}t_c + n\tau)$.

3.3 All-to-Some Personalized Communication

We only consider the case where $\mathcal{I} = \phi$ and $|\mathcal{R}_b| \neq |\mathcal{R}_a|$. If $|\mathcal{R}_b| = |\mathcal{R}_a| = |\mathcal{R}|$, then the communication is *all-to-all personalized communication.* The general case for which $\mathcal{I} \neq \phi$ is treated in [4]. If the number of real processor dimensions used before the transposition is greater than the number used after the transposition, i.e., $|\mathcal{R}_b| - |\mathcal{R}_a| = k > 0$, then the transposition implies k steps of *all-to-one personalized communication* and $|\mathcal{R}_a|$ steps of *all-to-all personalized communication.* Data accumulation takes place during the k steps of *all-to-one personalized communication.* If $|\mathcal{R}_a| - |\mathcal{R}_b| = k > 0$, then there are k steps of *one-to-all personalized communication* and $|\mathcal{R}_b|$ steps of *all-to-all personalized communication.* The k steps of *one-to-all personalized communication* implies data splitting.

Theorem 1 *The steps of* all-to-one *and* all-to-all personalized communication *used to realize* all-to-some personalized communication *can be performed in any order. Performing the* all-to-all personalized communication *first minimizes the data transfer time. For* some-to-all personalized communication *performing the* one-to-all personalized communication *first minimizes the data transfer time.*

Comm. capab.	Time
one-port	$T = (l\frac{PQ}{2^{k+l+1}} + \sum_{i=0}^{k-1} \frac{PQ}{2^{k+l-i}})t_c + (l\lceil\frac{PQ}{B_m 2^{k+l+1}}\rceil + \sum_{i=0}^{k-1}\lceil\frac{PQ}{B_m 2^{k+l-i}}\rceil)\tau$
n-port	$T = (\frac{PQ}{2^{k+l+1}} + \frac{1}{k}\sum_{i=0}^{k-1} \frac{PQ}{2^{k+l-i}})t_c + (l\lceil\frac{PQ}{lB_m 2^{k+l+1}}\rceil + \sum_{i=0}^{k-1}\lceil\frac{PQ}{kB_m 2^{k+l-i}}\rceil)\tau$

Table 3: Estimated communication time for *Some-to-all personalized communication.*

The theorem simply states that data accumulation shall be performed last and data splitting first. The theorem can be proved by considering the communication complexity of inserting the k steps all-to-one (one-to-all) personalized communication among the all-to-all personalized communication.

Let $k = ||\mathcal{R}_b| - |\mathcal{R}_a||$ and $l = \min(|\mathcal{R}_b|, |\mathcal{R}_a|)$. If the minimized algorithm is executed, for the k steps of *all-to-one* or *one-to-all personalized communication* there are 2^l distinct subcubes in which the operation takes place concurrently. Each such subcube is of dimension k. Also, the *all-to-all personalized communication* takes place within subcubes of dimension l, and there are 2^k such subcubes.

The complexity estimates for $k = ||\mathcal{R}_b| - |\mathcal{R}_a||$ steps of accumulation/splitting and $l = \min(|\mathcal{R}_b|, |\mathcal{R}_a|)$ steps of *all-to-all personalized communication* are given in Table 3. Note that $l = n$, $k = 0$ yields the complexity of the *all-to-all personalized communication*, and $l = 0$, $k = n$ yields the complexity of the *one-to-all* or *all-to-one personalized communication.* In general, it is a 2^l-to-2^{l+k} (or 2^{l+k}-to-2^l) personalized communication.

4 Matrix Transposition

We have defined matrix transposition as a set of shuffle operations. This definition is convenient on certain processor networks, and for parts of the analysis. Matrix transposition implies an exchange of the row and column address fields. This exchange can clearly be accomplished by the *standard exchange algorithm*, if $p = q$. If this is not the case, then virtual elements can be introduced to square up the matrix. A standard exchange algorithm can be formulated as follows:

```
For $i := q - 1$ downto 0
    forall $u_i \oplus v_i = 1$
        exchange elements $\{(u||v)\}$ and $\{(v||u)\}$;
    endforall
endfor
```

Lemma 5 *Let $p = q$, $u_j = v_j$, $\forall\ j \in \{0, 1, \ldots, i-1, i+1, \ldots, q-1\}$, $u_i = \overline{v_i}$, then $Hamming((u||v),\ (v||u)) = 2$.*

Corollary 4 *If the number of processors is equal to the number of matrix elements, 2^m, then matrix transposition performed through a sequence of exchanges requires $\frac{1}{2}m$ exchanges, each requiring communication over a distance of two.*

Corollary 4 gives an upper bound equal to the lower bound of Corollary 2.

With a one-dimensional partitioning of the matrix, $\mathcal{I} = \phi$ regardless of the assignment schemes before and after the transposition. In the two-dimensional partitioning $\mathcal{I}$ may be empty, but it can also be equal to the full processor set $\mathcal{R}$.

Lemma 6 *If the exchange algorithm is used for transposition and $g(i), f(i) \in \mathcal{R}_b$, then the communication is between real processors at distance 2. If $g(i) \in \mathcal{R}_b, f(i) \notin \mathcal{R}_b$, or $g(i) \notin \mathcal{R}_b, f(i) \in \mathcal{R}_b$, then the communication is between real processors at distance 1. Otherwise, the exchange operation is a local data movement.*

5 One-Dimensional Matrix Partitioning

If there are data elements for every real processor both before and after the data rearrangement, then the matrix transposition is *all-to-all personalized communication.* Each node sends $\frac{PQ}{N^2}$ elements to every other node. The communication is *all-to-all personalized communication* regardless of whether or not the scheme for assigning elements to processors is the same before or after the transposition.

If the exchange algorithm is used for *all-to-all personalized communication* then the exchange operations takes place either between a virtual processor and a real processor or two virtual processors in the same real processor. With the same number of processors being used before and after the transposition and *one-port* communication the exchange algorithm is optimum within a factor of 2.

For matrix transposition by the exchange algorithm [10] presented next it is assumed that the matrix is partitioned consecutively by rows and that processor i initially holds the elements of the i^{th} block row. After the transpose operation it shall hold the elements of the i^{th} block column. Note that the number of rows in a block row is different from the number of columns in a block column, unless $P = Q$. However, the number of elements in a block row and a block column are the same. For the transpose operation the block row of each processor is partitioned by columns into N same-sized blocks. The transpose is formed by processor i exchanging its j^{th} block with the i^{th} block of processor j. The data array in each processor holding the elements of a block row is two-dimensional, unless the number of rows is equal to the number of processors, and the local data array after the transposition is also two-dimensional, unless the number of columns is less than or equal to the number of processors. To complete the transposition after the interprocessor communication is completed, this two-dimensional data array can be transposed further locally, explicitly, or implicitly by indirect addressing.

```
/* Transposition by the Standard Exchange Algorithm: */
for $j := n - 1$ downto 0
    if (bit $j$ of my-addr $= 0$) then
        exchange blocks $\frac{1}{2}N$ to $N - 1$ of my blocked array
            with my neighbor in dimension $j$
    else
        exchange blocks 0 to $\frac{1}{2}N - 1$ of my blocked array
            with my neighbor in dimension $j$
    endif;
    shuffle my blocked array;
endfor
```

The loop can also be performed with the loop index running in the opposite order, but then the first operation in the loop shall be an unshuffle operation, which replaces the shuffle operation at the end of the loop.

For *n-port* communication the exchange algorithm is no longer optimal. A SBnT algorithm as described below yields a communication complexity that is optimum within a factor of 2.

```
/* Transposition by a SBnT Algorithm: */
/* Let the format of msg be (source-addr, relative-addr, data). */
/* base($j$) = the minimum number of right rotation of $j$ which yields */
/* the minimum value among all rotations of $j$. */
for all $j \neq$ my-addr do
    form msg for processor $j$ = (my-addr, my-addr $\oplus j \oplus 00..01_b0..0$, data)
        and append to output-buf [$b$] where $b$ is the base of my-addr $\oplus j$.
loop $n$ times
    send concurrently for all $n$ output ports.
    receive concurrently for all $n$ input ports.
    for each $j$ do, $0 \leq j < n$
        for each msg of input-buf [$j$] do
            if relative-addr $= 0$ then
                put the data into the source-addr$^{th}$ block
                    of the target array
            else
                form relative-addr := relative-addr $\oplus$ $(0..01_p0..0)$ in
                the *msg* and append to output-buf [$p$], where $p$ is
                the bit position of *relative-addr* which is the
                nearest 1-bit to the left of the $j^{th}$ bit cyclically.
            /* Note: $j^{th}$ bit is always 0 here. */
        endif
    endfor
  endfor
endloop
```

In the case where only a few processing nodes contain data before or after the transformation it is of the form *some-to-all* or *all-to-some personalized communication.* In the extreme case it is *one-to-all* or *all-to-one personalized communication. Virtual elements* can be introduced such that the same number of real processors are used before and after the transposition. Real processors with virtual elements participate in the exchange operations by receiving data. Virtual elements need not be communicated. The number of real elements being communicated in an exchange operation is not constant, in general, when virtual elements are introduced.

Corollary 5 *In a one-dimensional partitioning such that $|\mathcal{R}_b| = |\mathcal{R}_a|$ there exist elements that must traverse $|\mathcal{R}_b|$ dimensions.*

The communication complexity for these cases are summarized in Table 3.

Lemma 7 *One-dimensional transposition can be combined with change of data assignment scheme in using the standard exchange algorithm.*

Corollary 6 *The conversion of the storage form of a matrix stored in $2^{|\mathcal{R}_b|} \leq 2^n$ processors from any one of the following storage forms*

- *consecutive row*
- *consecutive column*
- *cyclic row*
- *cyclic column*
- *combined cyclic and consecutive row storage*
- *combined cyclic and consecutive column storage*

to any other of these forms requires communication from each of the processors to $2^{|\mathcal{R}_a|} - 1$ *other processors, if* $I = \phi$.

Corollary 7 *The conversion between the cyclic and consecutive storage forms implies* all-to-all personalized communication, *if* $P \geq N^2$ *for partitioning by rows and* $Q \geq N^2$ *for partitioning by columns.*

For both the SBT and SBnT algorithms presented above it is assumed that the partitions are embedded in the cube by a binary encoding. For Gray code encoding of partitions and binary encoding of virtual processors, we can first perform a transformation locally such that block w is moved to block location $G(w)$, then carry out the above algorithms. The two operations can also be combined as described in section 6.2.

6 Two-Dimensional Partitioning

In the two-dimensional partitioning with cyclic assignment and the same number of dimensions for rows and columns the address field is partitioned as follows:

$$(\underbrace{u_{p-1}u_{p-2}\dots u_{n_c}}_{vp}\underbrace{u_{n_c-1}\dots u_0}_{rp}\ \underbrace{v_{q-1}v_{q-2}\dots v_{n_c}}_{vp}\underbrace{v_{n_c-1}\dots v_0}_{rp}).$$

For consecutive assignment the partitioning is

$$(\underbrace{u_{p-1}u_{p-2}\dots u_{p-n_c}}_{rp}\underbrace{u_{p-n_c-1}\dots u_0}_{vp}\ \underbrace{v_{q-1}v_{q-2}\dots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}\dots v_0}_{vp}).$$

In either of these cases $I = \mathcal{R}_b = \mathcal{R}_a$. This case is the basic two-dimensional matrix transposition. The communication is between pairs of processors. In [9,10] we show that the transposition of a matrix embedded in the cube by a binary code or Gray code encoding implies the same communication. The algorithm in the above references uses a single path from source to destination for every source/destination pair. We will describe a simple extension using two paths for every source/destination pair, and an algorithm using multiple paths. We refer to the three algorithms by the names *Single Path Transpose* (SPT), *Dual Path Transpose* (DPT), and *Multiple Path Transpose* (MPT).

Note that by corollary 2 the maximum distance matrix elements need to traverse is $n = 2n_c$.

With a mixed assignment before and after the transposition, such as consecutive for rows and cyclic for columns

$$(\underbrace{u_{p-1}u_{p-2}\dots u_{p-n_r}}_{rp}\underbrace{u_{p-n_r-1}\dots u_0}_{vp}\ \underbrace{v_{q-1}v_{q-2}\dots v_{n_c}}_{vp}\underbrace{v_{n_c-1}\dots v_0}_{rp}),$$

the set I may no longer equal the entire processor set. In fact, if $q - n_c \geq n_r$ and $p - n_r \geq n_c$ then $I = \phi$ and it is an *all-to-all personalized communication.* Between these two extremes the communication is discussed in [4].

6.1 Transposition with Communication Only between Distinct Source/Destination Pairs of Processors

We consider the transpose operation for binary encoding first. Define $tr(x)$ to be the function which maps the address of partition (address of assigned processor) $x = (x_r||x_c)$ to the address of the transposed partition, i.e., $tr(x) = (x_c||x_r)$. Let $H(x) = Hamming(x_r, x_c)$. Then $Hamming(x, tr(x)) = 2H(x)$. In the following we assume that $n_r = n_c = \frac{n}{2}$ and n is even, i.e., that there are equally many row and column partitions.

The *Single Path Transpose* (SPT) algorithm [10,15] uses one path from processor x to processor $tr(x)$. Paths for different $x's$ are edge-disjoint, and pipelining of communications can be employed to reduce the communication complexity. The *Dual Paths Transpose* (DPT) algorithm is a straightforward improvement of the SPT algorithm in that two directed edge-disjoint paths are established from each source processor to its corresponding destination processor. In the *Multiple Paths Transpose* (MPT) algorithm, we partition the processor addresses into sets such that all members of a set have equivalent properties with respect to an relation operator (defined later). We show that the paths associated with any two source processors belonging to different sets are edge-disjoint. We then prove that all the paths of the processors in the same set share the same set of edges, but use them during different cycles. An algorithm similar to the MPT algorithm was also derived independently by Stout et al. [20,21].

6.1.1 The Single Path Transpose (SPT) Algorithm

With the same assignment scheme for rows and columns, and the same number of processors assigned to rows and columns, $n_c = n_r = \frac{n}{2}$ (n must be even) the communication is restricted to distinct source/destination pairs. The *Single Path Transpose* (SPT) algorithm [10,15] is a special case of the standard exchange algorithm.

Lemma 8 *In a two-dimensional partitioning such that the same number of dimensions are used for real processor addresses before and after the transposition and the same assignment scheme used before and after the transposition there exist elements that must traverse $rp = 2n_c$ dimensions.*

In the SPT algorithm for the same number of real processors for rows and columns, and the same assignment scheme for both rows and columns both before and after the transposition, data is exchanged between processors with addresses that differ in dimensions $g(i)$, $g(i) \in \mathcal{R}, g(i) < q$, and $f(i)$, $f(i) \in \mathcal{R}, q \leq f(i) < m$ in the i^{th} exchange step. The implied routing corresponds to directed edge-disjoint paths from each node x to $tr(x)$. For each source-destination pair there is a single path. This path only goes through the appropriate dimensions of the real processor addresses corresponding to the bits of x that need to be complemented to become the destination real address $tr(x)$. The routing order for the dimensions is the same for all nodes, for instance highest to lowest order for both row and column encoding, i.e., $g(\frac{n}{2}-1)$, $f(\frac{n}{2}-1)$, $g(\frac{n}{2}-2)$, $f(\frac{n}{2}-2)$, $\ldots$, $g(0)$, $f(0)$. The length of the path of node x is $2H(x)$. The first packet for each node on the anti-diagonal arrives after n routing steps and additional packets every cycle thereafter. The total number of routing steps is $\lceil \frac{PQ}{BN} \rceil + n - 1$. The nodes which are not on the anti-diagonal can either finish the transposition earlier in a "greedy" manner, or synchronize with the anti-diagonal nodes, i.e., the packet with the same ordinal number of all the nodes uses the same dimension (or idles) during the same step. The total transposition time T is $(\lceil \frac{PQ}{BN} \rceil + n - 1)(Bt_c + \tau)$. The optimal packet size, B_{opt}, is $\sqrt{\frac{PQ\tau}{N(n-1)t_c}}$ and the minimum time, $T_{min} = (\sqrt{\frac{PQ}{N}t_c} + \sqrt{(n-1)\tau})^2$.

6.1.2 The Dual Paths Transpose (DPT) Algorithm

The SPT algorithm can be improved by establishing two directed edge-disjoint paths between x and $tr(x)$ for all $x's$. In addition to the paths used in the SPT algorithm, a second path is defined by permuting processor row and column dimensions pairwise to yield a routing order selected from $f(\frac{n}{2}-1)$, $g(\frac{n}{2}-1)$, $f(\frac{n}{2}-2)$, $g(\frac{n}{2}-2)$, $\ldots$, $f(0)$, $g(0)$. The two directed paths for a particular x are edge-disjoint (as observed in [11] for the solution of tridiagonal systems on Boolean cubes). Moreover, the two directed paths for any x are edge-disjoint with respect to all paths for other $x's$. This second path can be used to reduce the time for data transfer by splitting the set of data $\frac{PQ}{N}$ into two equal parts. The path lengths are already minimal in the SPT algorithm. The communication complexity is $(\lceil \frac{PQ}{2BN} \rceil + n - 1)(Bt_c + \tau)$, which is minimized for $B = B_{opt} = \sqrt{\frac{PQ\tau}{2N(n-1)t_c}}$ and $T_{min} = (\sqrt{\frac{PQ}{2N}t_c} + \sqrt{(n-1)\tau})^2$. The speedup is approximately 2 for $\frac{PQ}{N}t_c \gg n\tau$, i.e., for Boolean cubes small relative to the problem size. Note that for the SPT algorithm it suffices that each node supports a total of n concurrent send or receive operations, whereas for the DPT algorithm n send operations concurrently with n receive operations are required for each node. Uni-directional communication suffices for the SPT algorithm, but bi-directional communication is required for the DPT algorithm.

6.1.3 The Multiple Paths Transpose (MPT) Algorithm

For the *Multiple Paths Transpose* (MPT) algorithm we define $2H(x)$ paths, labeled $0, 1, \ldots, 2H(x)-1$, between nodes x and $tr(x)$. The paths differ in the order in which the dimensions are routed. All paths originated from the same node have the same length.

Let $\alpha_{H(x)-1}, \alpha_{H(x)-2}, \ldots, \alpha_0, \beta_{H(x)-1}, \beta_{H(x)-2}, \ldots, \beta_0$ be the sequence of dimensions that need to be routed in descending order. We describe a path as a sequence of dimensions.

$$path\ p = \begin{cases} \alpha_{(p+H(x)-1)\bmod H(x)}, \beta_{(p+H(x)-1)\bmod H(x)}, \alpha_{(p+H(x)-2)\bmod H(x)}, \beta_{(p+H(x)-2)\bmod H(x)}, \ldots, \alpha_p, \beta_p. \\ \qquad \forall p \in \{0, 1, \ldots, H(x)-1\} \\ \beta_{(j+H(x)-1)\bmod H(x)}, \alpha_{(j+H(x)-1)\bmod H(x)}, \beta_{(j+H(x)-2)\bmod H(x)}, \alpha_{(j+H(x)-2)\bmod H(x)}, \ldots, \beta_j, \alpha_j. \\ \qquad j = p - H(x), \forall p \in \{H(x), H(x)+1, \ldots, 2H(x)-1\} \end{cases}$$

For example, if $x = (1001||0100)$, then $x_r = 1001, x_c = 0100, H(x) = 3$ and $tr(x) = (x_c||x_r) = (0100||1001)$. The distance between x and $tr(x)$ is 6. The 6 paths are defined as follows:

$path\ 0 = 7, 3, 6, 2, 4, 0.$ $\qquad$ $path\ 3 = 3, 7, 2, 6, 0, 4.$
$path\ 1 = 4, 0, 7, 3, 6, 2.$ $\qquad$ $path\ 4 = 0, 4, 3, 7, 2, 6.$
$path\ 2 = 6, 2, 4, 0, 7, 3.$ $\qquad$ $path\ 5 = 2, 6, 0, 4, 3, 7.$

Path 0 starts from the source node (10010100) and goes through nodes (00010100), (00011100), (01011100), (01011000), (01001000) and reaches the destination node (01001001). Path p can be derived by a right rotation of two steps of path $(p-1) \bmod H(x)$, if $0 \leq p < H(x)$. For $H(x) \leq p < 2H(x)$, path p can be derived by a right rotation of two steps of path $((p-1) \bmod H(x)) + H(x)$ and also by permuting row and column dimensions pairwise of path $p \bmod H(x)$. Note that path 0 is the same as the path defined in the SPT algorithm. Paths 0 and $H(x)$ are the two paths defined for node x in the DPT algorithm.

Definition 12 *Let x', x'' be two nodes with $x' = (x'_r||x'_c)$ and $x'' = (x''_r||x''_c)$. Define a relation $\sim_{ad}$ between x' and x'' such that $x' \sim_{ad} x''$ iff $x'_r + x'_c = x''_r + x''_c$, i.e., x' and x'' are on the same anti-diagonal. Note that if $x' \sim_{ad} x''$ and $x'' \sim_{ad} x'''$ then $x' \sim_{ad} x'''$.*

Definition 13 *Define $edge(x, p, e)$ to be the function which returns the e^{th} directed edge of path p of node x, with $e \geq 1$. We also define Edges, OddEdges, EvenEdges and Paths as follows.*

$$Edges(x, e) = \{edge(x, p, e) | \forall p \in \{0, 1, \ldots, 2H(x)-1\}\},$$

$$OddEdges(x) = \bigcup_{\forall\ odd\ e} Edges(x, e),$$

$$EvenEdges(x) = \bigcup_{\forall\ even\ e} Edges(x, e),$$

$$Paths(x) = OddEdges(x) \bigcup EvenEdges(x).$$

Definition 14 *Define $Nodes(x, e)$ to be the function which returns the set of nodes upon which the directed edges in $Edges(x, e)$ terminate. Define $OddNodes(x)$ and*

EvenNodes(x) to be the set of nodes on which the set of directed edges OddEdges(x) and EvenEdges(x) terminate, respectively.

$$OddNodes(x) = \bigcup_{\forall \; odd \; e} Nodes(x,e)$$

$$EvenNodes(x) = \bigcup_{\forall \; even \; e} Nodes(x,e)$$

Definition 15 *Let x', x'' be two nodes. Define a relation $\sim_s$ such that $x' \sim_s x''$ iff $x' \sim_{ad} x''$ and $x' \oplus tr(x') = x'' \oplus tr(x'')$. Note that if $x' \sim_s x''$ and $x'' \sim_s x'''$ then $x' \sim_s x'''$.*

$x' \oplus tr(x') = x'' \oplus tr(x'')$ implies $H(x') = H(x'')$, but $H(x') = H(x'')$ does not imply $x' \oplus tr(x') = x'' \oplus tr(x'')$. There exists x', x'' such that $x' \sim_{ad} x''$ and $x' \oplus tr(x') \neq x'' \oplus tr(x'')$, for instance $(001||111)$ and $(010||110)$. Also there exists x', x'' such that $x' \not\sim_{ad} x''$ and $x' \oplus tr(x') = x'' \oplus tr(x'')$, for instance $(001||111)$ and $(000||110)$.

Definition 16 *A set of paths defined upon a set of nodes $\mathcal{X}$ is said to be (t,n)-disjoint, $t \leq n$, if a packet that can be transmitted in unit time can be sent out on every path from every node $x \in \mathcal{X}$ during cycles $i*n+1$, $i*n+2$, $\ldots$, $i*n+t$, $\forall\, i \geq 0$, without routing conflicts, i.e., messages originating from different nodes will not be routed over the same edge during the same cycle.*

Note that the (t,n)-disjoint definition does not imply that the paths from the different source nodes are edge-disjoint, unless $t = n$.

To describe the MPT algorithm we first prove the following properties.

1. Paths p_1 and p_2 of node x are edge-disjoint, $\forall p_1, p_2 \in \{0, 1, \ldots, 2H(x)-1\}$, $p_1 \neq p_2$.
2. If $x' \not\sim_s x''$ then $Paths(x') \cap Paths(x'') = \phi$.
3. The set of all paths for the nodes in the set induced by the relation $\sim_s$ is $(2, 2H(x))$-disjoint where x is in the node set.

Lemma 9 *Paths p_1 and p_2 of node x are edge-disjoint, $\forall p_1, p_2 \in \{0, 1, \ldots, 2H(x)-1\}$, $p_1 \neq p_2$.*

Proof: It follows from the facts that all the paths are pointing away from the source node and no two paths traverse the same dimension during the same step. ▮

Lemma 10 *If $H(x') > 0$, then the set of nodes $OddNodes(x')$ and $EvenNodes(x')$ have the following properties:*

- $x' \not\sim_{ad} x''$, $H(x'') = H(x') - 1$, $\forall\ x'' \in OddNodes(x')$,
- $x' \sim_{ad} x''$, $x' \oplus tr(x') = x'' \oplus tr(x'')$ *(which implies* $H(x'') = H(x')$*),* $\forall\ x'' \in EvenNodes(x')$.

Proof: In traversing an edge in $OddEdges(x)$, we complement one of the $H(x)$ bits of the $\frac{n}{2}$ high (low) order bits which differ from the corresponding low (high) order bit. In traversing an edge in $EvenEdges(x)$, we complement the low (high) order bit of the corresponding high (low) order bit that was complemented in traversing the preceding odd edge. Let x', x'', and x''' be nodes along the same path such that $x' = (y'||z') \in Nodes(x, 2h)$, $x'' = (y''||z'') \in Nodes(x, 2h+1)$ and $x''' = (y'''||z''') \in Nodes(x, 2h+2)$, $\forall h \in \{0, 1, \ldots, H(x) - 1\}$. From the definition of paths either $y'' = y' + 2^k, z'' = z'$ or $y'' = y', z'' = z' - 2^k$ for some k satisfying $y'_k = 0$, $z'_k = 1$; or $y'' = y' - 2^k, z'' = z'$ or $y'' = y', z'' = z' + 2^k$ for some k satisfying $y'_k = 1, z'_k = 0$. These conditions imply $y' + z' \neq y'' + z''$,i.e., $x' \not\sim_{ad} x''$, and $Hamming(y'', z'') = Hamming(y', z') - 1$, i.e., $H(x'') = H(x') - 1$. Furthermore, $y''' = y' + 2^k, z''' = z' - 2^k$ for some k satisfying $y'_k = 0, z'_k = 1$ or $y''' = y' - 2^k, z''' = z' + 2^k$, for some k satisfying $y'_k = 1, z'_k = 0$. Hence, $y' + z' = y''' + z'''$, i.e., $x' \sim_{ad} x'''$. Also, $y' \oplus z' = y''' \oplus z'''$, i.e., $(y'||z') \oplus (z'||y') = (y'''||z''') \oplus (z'''||y''')$ which implies $x' \oplus tr(x') = x''' \oplus tr(x''')$. ▌

Corollary 8 $x' \sim_s x''$, $\forall\ x'' \in EvenNodes(x')$.

Lemma 11 *If $x' \not\sim_{ad} x''$, then $Paths(x') \cap Paths(x'') = \phi$.*

Proof: It is sufficient to prove $Paths(x') \cap Paths(x'') = \phi$ by proving $EvenNodes(x') \cap EvenNodes(x'') = \phi$ and $EvenNodes(x') \cap OddNodes(x'') = \phi$. From lemma 10, $EvenNodes(x') \sim_{ad} x'$, $EvenNodes(x'') \sim_{ad} x''$. Since $x' \not\sim_{ad} x''$, we have $EvenNodes(x') \not\sim_{ad} EvenNodes(x'')$, which implies $EvenNodes(x') \cap EvenNodes(x'') = \phi$.

To prove $EvenNodes(x') \cap OddNodes(x'') = \phi$, we consider three cases.

1. If $H(x') = H(x'')$, then by lemma 10 $H(y') = H(y'') + 1$ where $y' \in EvenNodes(x')$, $y'' \in OddNodes(x'')$. So, $EvenNodes(x') \cap OddNodes(x'') = \phi$.
2. If $H(x') > H(x'')$, then $H(y') > H(y'')$ where $y' \in EvenNodes(x')$, $y'' \in OddNodes(x'')$. So, $EvenNodes(x') \cap OddNodes(x'') = \phi$.
3. If $H(x') < H(x'')$, we show $EvenNodes(x'') \cap OddNodes(x') = \phi$ instead by a similar argument as in case 2. ▌

Lemma 12 *If $x' \sim_{ad} x''$ and $x' \not\sim_s x''$, then $Paths(x') \cap Paths(x'') = \phi$.*

Proof: Assume $EvenNodes(x') \cap EvenNodes(x'') \neq \phi$, then there exists one node y such that $y \in EvenNodes(x')$ and $y \in EvenNodes(x'')$. By corollary 8, $y \sim_s x', y \sim_s x''$, i.e., $x' \sim_s x''$ which is a contradiction. So, $EvenNodes(x') \cap EvenNodes(x'') = \phi$. Also by lemma 10, $y' \not\sim_{ad} y''$, $\forall y' \in EvenNodes(x')$ and $y'' \in OddNodes(x'')$, which means $EvenNodes(x') \cap OddNodes(x'') = \phi$. Hence, $Paths(x') \cap Paths(x'') = \phi$. ▮

Lemma 13 *If $x' \not\sim_s x''$ then $Paths(x') \cap Paths(x'') = \phi$.*

Proof: It follows from lemmas 11 and 12. ▮

Lemma 14 *The set of paths defined for the nodes in the same set induced by the relation $\sim_s$ is $(2, 2H(x))$-disjoint.*

Proof: We first prove that the paths of the nodes defined by the relation $\sim_s$ are $(1, 2H(x))$-disjoint. The proof is by induction on the routing cycles. During cycles 1 and 2, the routed edges are clearly disjoint by Lemma 10. Assume that during cycles $2n-1$ and $2n$, $n > 0$, the routing is also edge-disjoint. If $n = H(x)$, then all the routing is complete. During the next two cycles the routing is restarted and there is no edge conflict. If $n \neq H(x)$, then consider the $2H(x)$ edges directed into some node y at distance $2n$ from x as well as the $2H(x)$ edges directed out from node y. Let $\alpha_{H(x)-1}, \alpha_{H(x)-2}, \ldots, \alpha_0, \beta_{H(x)-1}, \beta_{H(x)-2}, \ldots, \beta_0$ be the corresponding $2H(x)$ dimensions in descending order. If an edge used during cycle $2n-1$ is in dimension α_k (i.e., the edge used during cycle $2n$ is in dimension β_k) then the edges used during the following two cycles are in dimensions $\alpha_{(k-1) \bmod H(x)}$ and $\beta_{(k-1) \bmod H(x)}$ respectively. If the edge used during cycle $2n-1$ is in dimension β_k then the edges used during the following two cycles are in dimensions $\beta_{(k-1) \bmod H(x)}$ and $\alpha_{(k-1) \bmod H(x)}$ respectively. Hence, the edges used during the following two cycles are all distinct and it follows that the paths are $(1, 2H(x))$ disjoint.

To show that the paths are $(2, 2H(x))$-disjoint it suffices to show that the set of edges used during odd cycles (odd edges) are disjoint from the set of edges used during even cycles (even edges). Let x be any node in the set defined by the relation $\sim_s$. That the set of edges used during odd cycles are disjoint from the set of edges used during even cycles follows from the property that odd edges are directed from node x' to node y' and even edges directed from node y'' to node x'' where $x \sim_s x' \sim_s x''$, $x \not\sim_s y'$ and $x \not\sim_s y''$. ▮

Figure 3 shows an instance of a set induced by the relation $\sim_s$ on a 6-cube. Note that $H(x) = 3$ for x in this set. The nodes in the same set form a *logical* $H(x)$-dimensional

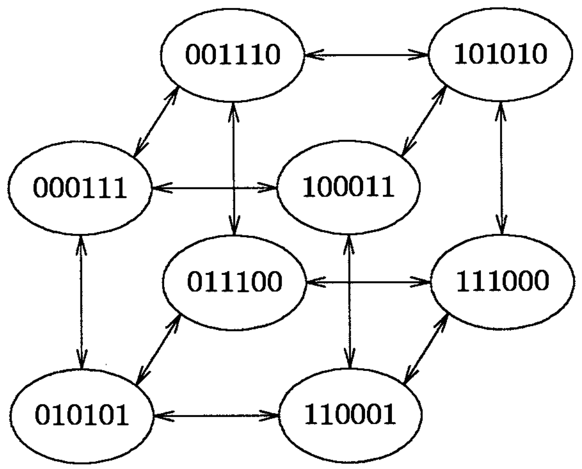


Figure 3: The *logical* $H(x)$-cube formed by nodes in the same induced set of the relation $\sim_s$.

cube where each logical link represents an exchange operation of two dimensions. Hence, a logical link contains two disjoint paths of length two. By lemma 13, the corresponding physical edges of the logical link will only be shared by nodes in this set. Notice that x and $tr(x)$ are at maximum distance from each other in the logical $H(x)$-cube. Figure 4 shows the 6 $(2H(x))$ edge-disjoint paths from node $x = (000111)$ to node $tr(x) = (111000)$. The labels on the edges are dimensions of the edges.

For the routing, the data from node x is split into $4H(x)$ packets of size $\lceil \frac{PQ}{4NH(x)} \rceil$ each. The packets are sent during the first two cycles. The first $2H(x)$ packets will arrive at the destination node, $tr(x)$, after $2H(x)$ cycles, and the second set during the next cycle. The total transpose time is

$$\begin{cases} (n+1)\tau + (\frac{n+1}{2n})\frac{PQ}{N}t_c & \text{if } \frac{n}{2} \geq \frac{PQt_c}{8N\tau}; \\ 3\tau + \frac{3}{4}\frac{PQ}{N}t_c & \text{otherwise.} \end{cases}$$

The transpose time decreases as a function of $H(x)$ for $1 \leq H(x) \leq \sqrt{\frac{PQt_c}{8N\tau}}$ and increases for $\sqrt{\frac{PQt_c}{8N\tau}} \leq H(x)$. The transpose time for $H(x) = 1$ and $H(x) = \frac{PQt_c}{8N\tau}$ are the same. The maximal packet size is $\frac{PQ}{4N}$. The maximal packet size can be reduced either without affecting the total transpose time (if $\frac{n}{2} \geq \frac{PQt_c}{8N\tau}$) or the total transpose time reduced by splitting the data into $\lfloor \frac{n}{2H(x)} \rfloor * 4H(x)$ packets. In fact, the data sent from node x can be split into $4kH(x)$ packets instead of $4H(x)$ packets. The whole routing completes in $2kH(x)+1$ cycles. Hence, $T = (2kH(x)+1)(\tau + \frac{PQt_c}{4kH(x)N})$, $H(x) \in \{1, 2, \ldots, \frac{1}{2}n\}$. The optimal k is $\frac{1}{2H(x)}\sqrt{\frac{PQt_c}{2N\tau}}$ and $T_{min} = (\sqrt{\tau} + \sqrt{\frac{PQt_c}{2N}})^2$. Notice that T_{min} is valid only when $k \geq 1$, which implies $\sqrt{\frac{PQt_c}{2N\tau}} \geq n$.

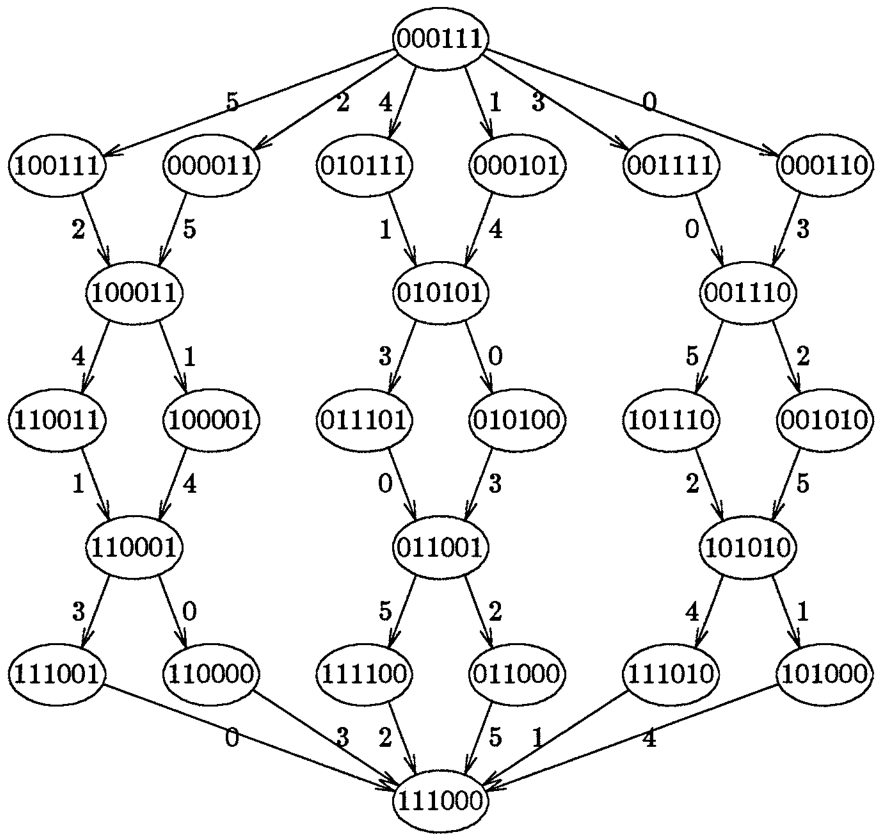


Figure 4: 6 ($2H(x)$) edge-disjoint paths from node $x = (000111)$ to node $tr(x) = (111000)$.

Theorem 2 *The total matrix transpose time by the MPT algorithm is*

$$T_{min} = \begin{cases} (n+1)\tau + \frac{n+1}{2n}\frac{PQ}{N}t_c & \text{if } n \geq \sqrt{\frac{PQt_c}{N\tau}} \text{ approximately;} \\ (\frac{n}{2}+3)\tau + \frac{n+6}{2n+8}\frac{PQ}{N}t_c & \text{if } \sqrt{\frac{PQt_c}{2N\tau}} < n \leq \sqrt{\frac{PQt_c}{N\tau}} \text{ approximately and } \frac{n}{2} \text{ is even;} \\ (\frac{n}{2}+2)\tau + \frac{n+4}{2n+4}\frac{PQ}{N}t_c & \text{if } \sqrt{\frac{PQt_c}{2N\tau}} < n \leq \sqrt{\frac{PQt_c}{N\tau}} \text{ approximately and } \frac{n}{2} \text{ is odd;} \\ (\sqrt{\tau} + \sqrt{\frac{PQt_c}{2N}})^2 & \text{if } n \leq \sqrt{\frac{PQt_c}{2N\tau}}, \end{cases}$$

and the optimum packet size is

$$B_{opt} = \begin{cases} \lceil \frac{PQ}{N(n+4)} \rceil & \text{for even } \frac{n}{2} \text{ and } n > \sqrt{\frac{PQt_c}{2N\tau}}; \\ \lceil \frac{PQ}{N(n+2)} \rceil & \text{for odd } \frac{n}{2} \text{ and } n > \sqrt{\frac{PQt_c}{2N\tau}}; \\ \sqrt{\frac{PQ\tau}{2Nt_c}} & \text{for } n \leq \sqrt{\frac{PQt_c}{2N\tau}}. \end{cases}$$

Theorem 3 *The matrix transposition time is at least* $\max(n\tau, \frac{PQ}{2N}t_c)$.

Proof: The minimum number of start-ups is determined by the longest distance, which is n. Nodes on the main anti-diagonal are at distance n. For a lower bound on the required time for data transfer consider the upper right $\frac{\sqrt{N}}{2} \times \frac{\sqrt{N}}{2}$ submatrix. There are $\frac{N}{4}$ nodes. Each node has to send $\frac{PQ}{N}$ data to some node outside the submatrix. There are two links per node that connects to nodes outside of the submatrix, i.e., a total of $\frac{2N}{4}$ links. Hence, the data transfer requires a time of at least $\frac{PQ}{2N}t_c$. ▮

For Gray code encoding on both row and column indices, we can apply exactly the same transpose algorithm. For a binary encoding of row and column indices, matrix element (u, v) is stored in processor $w = (u||v)$ and matrix element (v, u) is stored in processor $tr(w) = (v||u)$. For Gray code encoding of row and column indices, matrix element (u, v) is stored in processor $(G(u)||G(v))$ and matrix element (v, u) is stored in processor $(G(v)||G(u))$. The two-dimensional transpose algorithms described above are indeed permutation algorithms defined by $(u||v) \leftarrow (v||u), \forall u \in \{0, 1, \ldots, P-1\}, \forall v \in \{0, 1, \ldots, Q-1\}$. It follows that the permutation will transpose the matrix. In general, if row and column indices are encoded in the same way, the transpose algorithm only depends on the processor addresses, not on the row and column indices of the matrix elements in the processors. For $N < PQ$, the argument applies to matrix blocks instead of matrix elements.

6.2 Transposition with Change of Assignment Scheme

If the number of processors in the row and column direction are not the same, or if a different assignment strategy is used for rows and columns, or if the assignment scheme after the transpose is different from that before the transpose, then the communication is no longer confined to distinct pairs. If $|\mathcal{R}_b| = |\mathcal{R}_a| = |\mathcal{R}|$ and $\mathcal{I} = \phi$, then the communication is ***all-to-all personalized communication.*** In general, for $\mathcal{I} \neq \phi$ the transposition/rearrangement is composed of different types of operations. This case is treated further in [4].

For a non-square matrix virtual elements can be introduced. Virtual elements need not be communicated, and the complexity of the transposition is reduced accordingly, but the basic algorithms apply.

To illustrate a two-dimensional transposition with change of assignment scheme, such that $\mathcal{I} = \phi$, we consider the transposition of a matrix stored consecutively with respect to both rows and columns before the transposition, and stored cyclically with respect to both rows and columns after the transposition. We also assume that $n_r = n_c$ and that $p, q \geq 2n_r$. The partitioning of the address field before and after the transposition and change of assignment scheme are

$$\textit{Before}: \quad (\underbrace{u_{p-1}u_{p-2}\ldots u_{p-n_c}}_{rp}\underbrace{u_{p-n_c-1}\ldots u_0}_{vp}\;\underbrace{v_{q-1}v_{q-2}\ldots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}\ldots v_0}_{vp}),$$

$$\textit{After}: \quad (\underbrace{v_{q-1}v_{q-2}\ldots v_{n_c}}_{vp}\underbrace{v_{n_c-1}\ldots v_0}_{rp}\;\underbrace{u_{p-1}u_{p-2}\ldots u_{n_c}}_{vp}\underbrace{u_{n_c-1}\ldots u_0}_{rp}).$$

We consider three exchange algorithms that differ only in the way dimensions are paired, and the order in which the exchanges are performed. Let *exchange-row* (M, s, N_r) denote the sequence of exchange operations between N_r block rows (within a column subcube of N_r processors) as defined by the standard exchange algorithm described in pseudo code before, except for a minor modification. The initial local array of length M is partitioned into $2^s N_r$ blocks. The j^{th} block is sent to processor $j \bmod N_r$, $\forall j \in \{0, 1, \ldots, 2^s N_r - 1\}$ during the execution of the exchange algorithm. Each processor sends 2^s blocks to every other processor. For the exchange algorithm for the transposition of a one-dimensionally partitioned matrix described earlier, $M = \frac{PQ}{N}$, $s = 0$, $N_r = N$. Each processor sends only one block to every other processor. *Exchange-row* (M, s, N_r) operates within each column subcube. *Exchange-column* (M, s, N_c) is defined analogously.

The parameter s defines the offset from the high order dimension of the virtual processor address field for the first exchange in the ***standard exchange algorithm.*** From the discussion of the standard exchange algorithm it is clear that an offset of s divides the local array into 2^{s+1} blocks for the first exchange. The blocks are of size $\frac{PQ}{2^{s+1}N}$ for a $P \times Q$ matrix partioned evenly among N real processors.

For the transposition with change of assignment scheme we consider the following three algorithms:

1. Convert from consecutive-row partitioning to cyclic-row partitioning, i.e., *exchange-row* $(\frac{PQ}{N}, p - 2n_r, N_r)$; then convert from consecutive-column partitioning to cyclic-

column partitioning, by employing *exchange-column* $(\frac{PQ}{N}, m-n-n_c, N_c)$; then transpose the matrix globally and locally.

2. Transpose the local matrices concurrently; then convert from consecutive-row partitioning to cyclic-row partitioning, i.e., *exchange-row* $(\frac{PQ}{N}, p-2n_r, N_r)$; then convert from consecutive-column partitioning to cyclic-column partitioning, i.e., *exchange-column* $(\frac{PQ}{N}, m-n-n_c, N_c)$; then transpose N local matrices each of size $2^{p-2n_r} \times 2^{q-2n_c}$ concurrently in all N real processors.

3. Convert from consecutive-column to cyclic-column partitioning between rows (within each column subcube), i.e., *exchange-row* $(\frac{PQ}{N}, m-n-n_c, N_r)$; then convert from consecutive-row to cyclic-row partitioning between columns (within each row subcube), i.e., *exchange-column* $(\frac{PQ}{N}, p-n, N_c)$. A local $p-2n_r$ shuffle operation is necessary if $p > 2n_r$.

The algorithms can be illustrated in terms of operations on the address field. For simplicity let it be partitioned as $(u_1u_2u_3v_1v_2v_3)$, where u_1, u_3, v_1, and v_3 all define subfields of n_r dimensions. u_1 and v_1 are the real processor address fields before the transposition, u_3 and v_3 the real fields after transposition and change of assignment scheme.

Algorithm 1:

$$(\underline{u_1}u_2u_3\underline{v_1}v_2v_3) \rightarrow (u_1u_2\underline{u_3v_1}v_2v_3) \rightarrow (u_1u_2\underline{u_3}v_1v_2\underline{v_3}) \rightarrow (v_1v_2\underline{v_3}u_1u_2\underline{u_3}).$$

Algorithm 2:

$$(\underline{u_1}u_2u_3\underline{v_1}v_2v_3) \rightarrow (\underline{u_1}v_2v_3\underline{v_1}u_2u_3) \rightarrow (u_1v_2\underline{v_3v_1}u_2u_3) \rightarrow (u_1v_2\underline{v_3}v_1u_2\underline{u_3}) \rightarrow (v_1v_2\underline{v_3}u_1u_2\underline{u_3}).$$

Algorithm 3:

$$(\underline{u_1}u_2u_3\underline{v_1}v_2v_3) \rightarrow (\underline{v_3}u_2u_3\underline{v_1}v_2u_1) \rightarrow (\underline{v_3}u_2v_1\underline{u_3}v_2u_1) \rightarrow (\underline{v_3}v_1v_2\underline{u_3}u_1u_2).$$

The underline denotes the real processor address field. Note that the last form in algorithm 3, $(\underline{v_3}v_1v_2\underline{u_3}u_1u_2)$, denotes the same assignment scheme as $(v_1v_2\underline{v_3}u_1u_2\underline{u_3})$. The steps of the three different algorithms are illustrated in Figure 5 in terms of the matrix. The number in the Figure denotes (row-index||column-index).

The first algorithm requires $2n$ communication steps, the second only n steps. However, the second algorithm requires a complete local matrix transpose before the interprocessor communication phase, and the transposition of a number of smaller matrices after the communication. The third algorithm also requires n communication steps, but no transposition is required prior to the communication. A local $p-2n_r$ shuffle operation is required if $p > 2n_r$. Note that the order between exchange-row and exchange-column operations can be reversed.

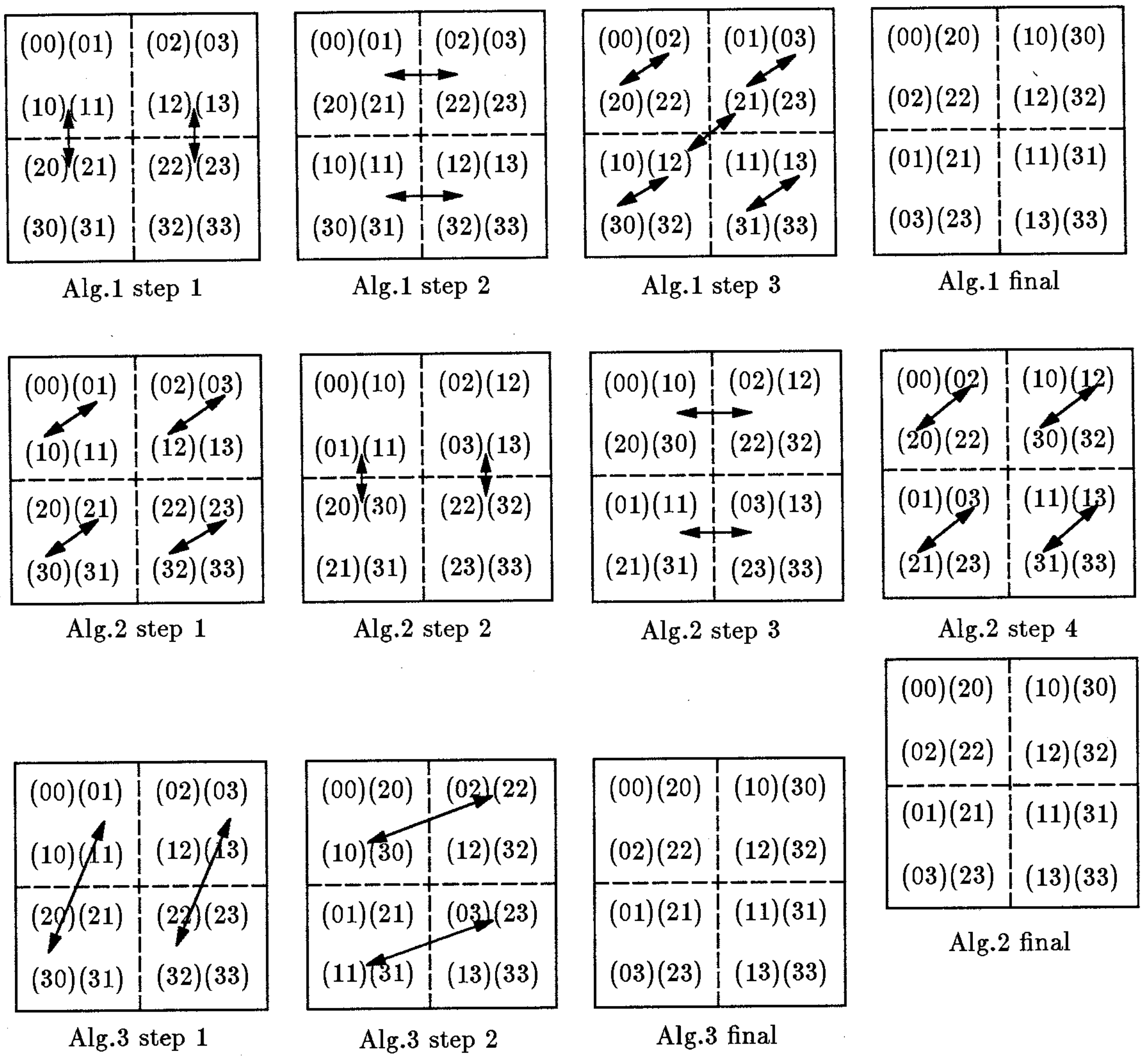


Figure 5: Three different algorithms to transpose a matrix from two-dimensional consecutive partitioning to two-dimensional cyclic partitioning.

Conversion between cyclic and consecutive assignment in the row or column direction is equivalent to a number (N_c or N_r) of independent one-dimensional conversions. Conversion in both dimensions is equivalent to *all-to-all personalized communication* if $Q \geq N_c^2$ and $P \geq N_r^2$.

6.3 Combining Transpose and Gray Code/Binary Code Conversion

For the transpose of a matrix with the row index encoded in binary code and the column index in Gray code, a binary-to-Gray-code conversion can first be done for each column subcube concurrently in $\frac{n}{2} - 1$ steps [10], then the Gray-to-binary-code conversion for each row subcube concurrently in another $\frac{n}{2} - 1$ steps followed by the n-step transpose algorithm. The two conversions and the transposition commute. The total number of routing steps is $2n - 2$. However, the number of routing steps can be reduced to n, if the SPT algorithm is used for the transposition by combining it with the conversion operations. Pipelining can be applied. For simplicity, we describe the non-pipelined version. As for the SPT algorithm, the combined algorithm is composed of $\frac{n}{2}$ iterations. Each iteration contains two routing steps. In iteration $i \in \{0, 1, \ldots, \frac{n}{2} - 1\}$, bits $\frac{n}{2} - i - 1$ of the row and column indices are changed by sending data through the corresponding dimensions. With the rows encoded in binary code and the columns in Gray code, matrix block (u, v) is stored in processor $(u||G(v))$ and matrix block (v, u) is stored in processor $(v||G(u))$. The direct transpose permutation is defined by exchanging data between processor $(u||G(v))$ and processor $(G^{-1}(G(v))||G(u))$, where $G^{-1}()$ is the inverse Gray code.

During the first iteration, the upper right block $(0x_{n-2}x_{n-3} \ldots x_{\frac{n}{2}}||1x_{\frac{n}{2}-2}x_{\frac{n}{2}-3} \ldots x_0)$ and the lower left block $(1x_{n-2}x_{n-3} \ldots x_{\frac{n}{2}}||0x_{\frac{n}{2}-2}x_{\frac{n}{2}-3} \ldots x_0)$ are exchanged in two steps. Neither row nor column conversions for the two encodings affects iteration 0, because the Gray and binary codes have identical most significant bits. During the second iteration, the Gray code encoding of the column indices forces a horizontal exchange within the blocks for the second half of the block rows. The binary code encoding of the row indices forces a vertical exchange for the second half of the block columns. The transpose operation requires an anti-diagonal exchange within all four blocks. The combined permutation pattern is shown in figure 6.

In general, the Gray code encoding of the columns causes a horizontal exchange within all the odd block rows with block rows numbered from 0. The binary code encoding causes a vertical exchange within all i^{th} block columns such that the parity of the binary encoding of i is odd. This can be proved from the conversion from binary code to Gray code proceeding from the most significant bit to the least significant bit (instead of a "low order to high order bit" conversion sequence[10]). Figure 7 shows the four iterations with $n = 8$, in which c means *clockwise rotation* and cc means *counterclockwise rotation.* The algorithm is presented below.

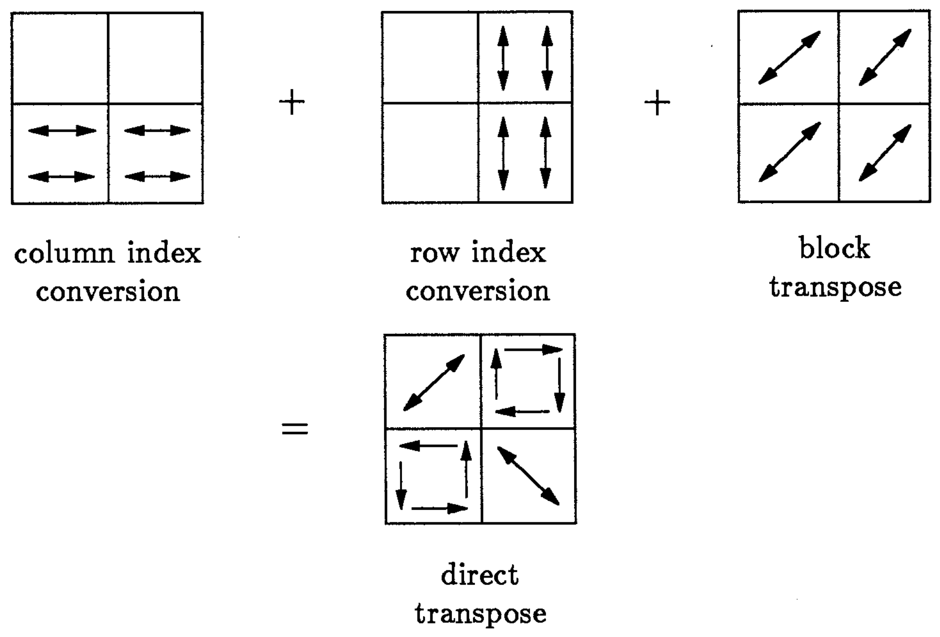


Figure 6: Transpose of a matrix stored by binary code encoding of row index and Gray code encoding of column index.

```
/* The second argument of "send" and "recv" represents the cube dimension */
/* and 'buf' contains the data to be transposed initially. */
even-block-row := true;
even-parity-block-column := true;
for $j := \frac{n}{2} - 1$ downto 0 do
    case (even-block-row, even-parity-block-column, bit $j + \frac{n}{2}$, bit $j$) of
        (TT00), (TT11), (FF01), (FF10):
            recv (tmp, $j + \frac{n}{2}$); send (tmp, $j$);
        (TT01), (TT10), (FF00), (FF11), (TF01), (TF10), (FT00), (FT11):
            send (buf, $j + \frac{n}{2}$); recv (buf, $j$);
        (TF00), (TF11), (FT01), (FT10):
            send (buf, $j$); recv (buf, $j + \frac{n}{2}$);
    endcase
    even-block-row := (bit $j + \frac{n}{2} = 0$);
    if (bit $j = 1$) then
        even-parity-block-column := not even-parity-block-column;
    endif
endfor
```

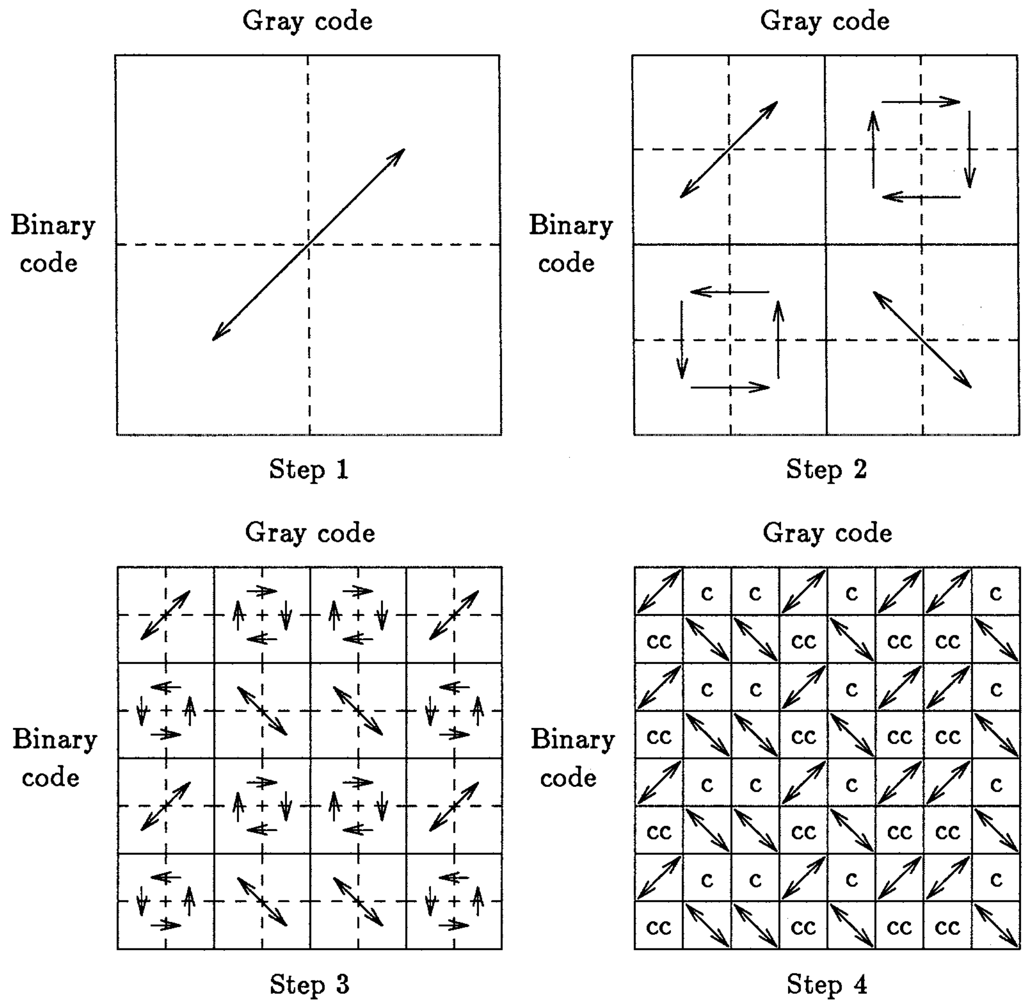


Figure 7: Transpose of a matrix stored by mixed encoding of rows and columns in an 8-cube.

The above algorithm was implemented on the Intel iPSC. The results are shown in Figure 15 in section 8 discussing experiments.

To transpose a matrix stored by binary encoding of row and column indices into a transposed matrix with row and columns encoded in Gray code, a combined conversion–transpose algorithm similar to the one above can be applied to accomplish the task in n routing steps. The algorithm above needs only be modified such that the column operations are controlled by even-block-columns (instead of even-parity-block-columns). Similarly, to transpose a matrix with both row and columns encoded in Gray code into a transposed matrix with rows and columns encoded in binary code, the control of the row operations is changed from even-block-rows to even-parity-block-rows.

7 Using Matrix Transposition for Other Permutations

For $I = \phi$, and $|\mathcal{R}_b| = |\mathcal{R}_a| = n$, matrix transposition is an *all-to-all personalized communication.* An arbitrary permutation on an n-cube can be realized by all-to-all personalized communication twice, if the size of messages to be permuted is the same for all processors and at least N (per processor) [21,20]. Since transposing a matrix with two-dimensional partitioning and $n_c = n_r$ is a permutation, one can also realize it by performing all-to-all personalized communication twice. However, the communication complexity is higher than that of the best transpose algorithm for the two-dimensional partitioning either for *one-port* communication, or for *n-port* communication.

The correspondence between cube dimensions for the *standard* exchange algorithm applied to matrix transposition is $f(i) = i$, $g(i) = i + \frac{n}{2}$, $\forall i \in \{0, 1, \ldots, \frac{n}{2} - 1\}$. By changing the exchange dimensions such that $f(i) = i$, $g(i) = n-1-i$, $\forall i \in \{0, 1, \ldots, \frac{n}{2} - 1\}$, a *bit-reversal* permutation is realized by the *general* exchange algorithm. A bit-reversal permutation is defined by

$$(x_{n-1}x_{n-2}\ldots x_0) \leftarrow (x_0 x_1 \ldots x_{n-1}).$$

Definition 17 *Define* dimension permutation *to be a permutation such that processor* $(x_{n-1}x_{n-2}\ldots x_0)$ *sends its data to processor* $(x_{\delta(n-1)}x_{\delta(n-2)}\ldots x_{\delta(0)})$ *where* δ *is a* $\{0, 1, \ldots, n-1\}$ *to* $\{0, 1, \ldots, n-1\}$ *permutation function.*

Definition 18 *Define* parallel swapping *to be a dimension permutation such that the permutation function* δ *satisfies* $\delta(\delta(i)) = i$, *i.e., either* $\delta(i) = i$ *or* $\delta(i) = j$, $\delta(j) = i$, $i \neq j$, $\forall i \in \{0, 1, \ldots, n-1\}$

Lemma 15 *Any* dimension permutation *can be realized by performing* parallel swapping $\lceil \log_2 n \rceil$ *times (note that n is the number of dimensions).*

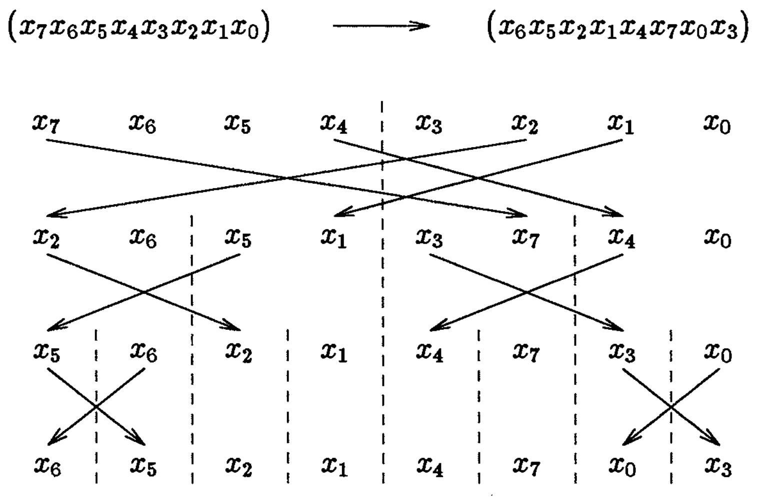

$(x_7x_6x_5x_4x_3x_2x_1x_0)$
$(x_6x_5x_2x_1x_4x_7x_0x_3)$
x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
x_2 x_6 x_5 x_1 x_3 x_7 x_4 x_0
x_5 x_6 x_2 x_1 x_4 x_7 x_3 x_0
x_6 x_5 x_2 x_1 x_4 x_7 x_0 x_3

Figure 8: Realizing *dimension permutation* by performing $\log n$ steps *parallel swapping.*

Proof: Assuming n is a power of two first. Arbitrarily partition the set of dimensions into two same-sized subsets, called S_1 and S_2 respectively. Let k be the cardinality of the set $\{i|i \in S_1, \delta(i) \in S_2\}$. Clearly, the cardinality of the set $\{i|i \in S_2, \delta(i) \in S_1\}$ is also k. Exchanging the k dimensions in S_1 with the corresponding k dimensions in S_2 can be done in one *parallel swapping* step. After this parallel swapping, there are two same-sized subsets which only require internal permutation. This permutation can be performed concurrently for the two subsets. Therefore, $\log n$ steps of *parallel swapping* suffice to realize the dimension permutation. For arbitrary n, we can add virtual elements such that the number of dimensions in the address field becomes a power of two. ∎

Figure 8 shows an example of permuting 8 dimensions by 3 steps of *parallel swappings.* Notice that k shuffle/unshuffle operations (left/right rotation k steps) fall in the *dimension permutation* class. There are $n!$ possible dimension permutations among $N!$ arbitrary permutations.

8 Experiments and Implementation Issues

8.1 One-Dimensional Partitioning

The Intel iPSC effectively allows communication on only one port at a time. Hence, we choose to implement the one-dimensional transpose using the exchange algorithm. In our implementation we do not perform local shuffle operations in order to arrange the data to be exchanged into one block for the sake of reducing the number of start-ups, since the copying time on the Intel iPSC is significant. Copying 1024 single precision

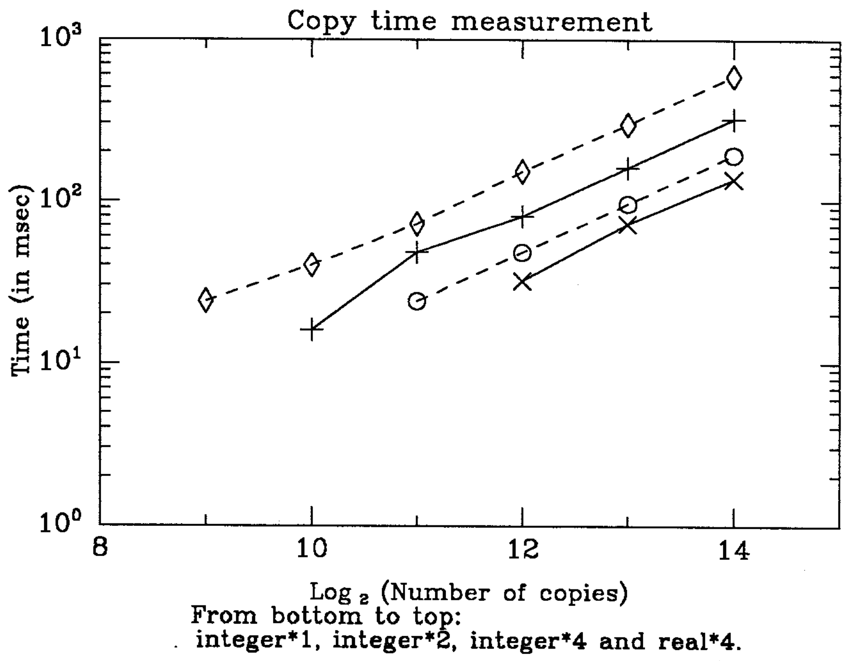


Figure 9: Measured times for copy of various data types on the Intel iPSC.

floating-point numbers ($4k$ *bytes*) takes about 37 milliseconds according to our measurements, Figure 9. The local array is partitioned into 2^j same-sized blocks during step j of the exchange algorithm. The odd or even blocks can either be sent directly to minimize the copy time, or copied into a buffer to reduce the number of start-ups. Figure 10 presents the measurements for unbuffered and buffered communication for rearrangement of consecutive to cyclic partitioning.

The complexity of the unbuffered communication is easily found to be $T = n\frac{PQ}{2N}t_c + (N + \lceil\frac{PQ}{2B_mN}\rceil \min(n, \log_2\lceil\frac{PQ}{B_mN}\rceil) - \frac{PQ}{B_mN})\tau$. With buffered communication, messages may initially be larger than the buffer size, in which case they are sent directly. Small messages are buffered and the time for communication is $T = n\frac{PQ}{2N}t_c + \frac{PQ}{N}\max(0, n - \log\lceil\frac{PQ}{B_{copy}N}\rceil)t_{copy} + (\min(N, \frac{PQ}{B_{copy}N}) - \min(N, \frac{PQ}{B_mN}) + \lceil\frac{PQ}{2B_mN}\rceil(\min(n, \log\lceil\frac{PQ}{B_mN}\rceil) + \max(0, n - \log\lceil\frac{PQ}{B_{copy}N}\rceil)))\tau$, where B_{copy} is the array size beyond which it is preferable with respect to performance to send without copying into a buffer. The complexity of the unbuffered communication grows linearly in the number of processors, i.e., exponentially in the number of cube dimensions, as shown in Figure 10. The buffered communication grows linearly in the number of cube dimensions. For a low growth rate it is important to have a large buffer, to reduce the number of start-ups, and fast copy. With the times for copy of floating-point numbers and communication start-ups on the Intel iPSC the copy of 64 single-precision floating-point numbers (256 bytes) takes approximately the same time as one communication start-up. Hence, it is beneficial with respect to performance

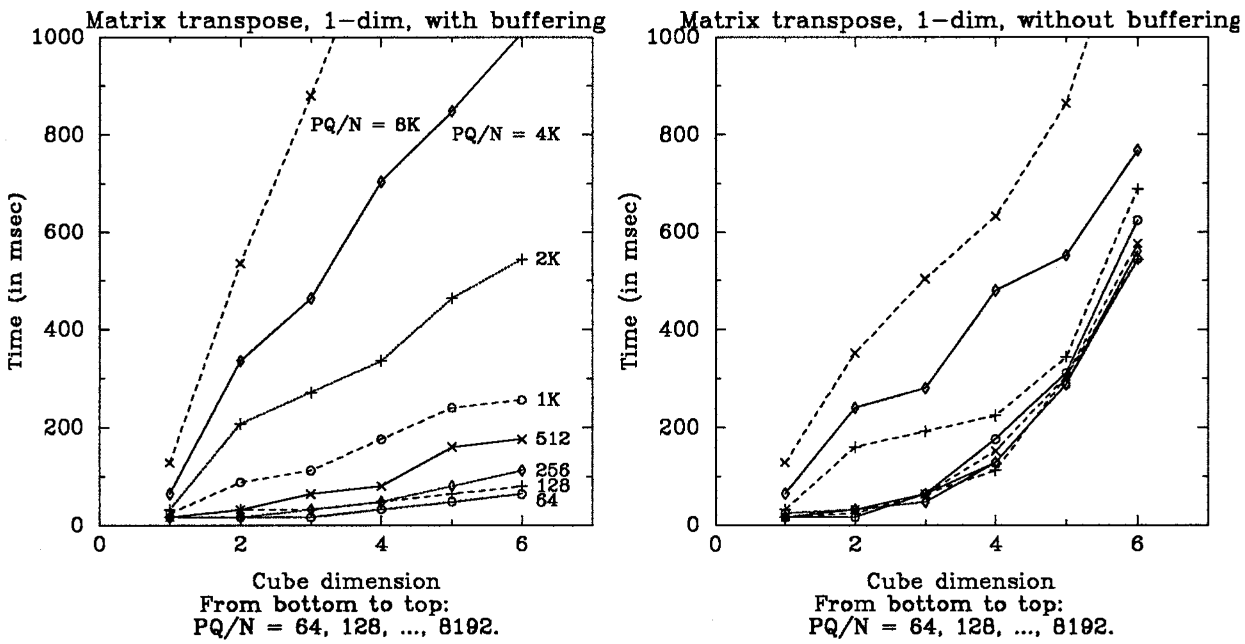


Figure 10: Measured times on the Intel iPSC for the transpose of a matrix, one-dimensional partitioning (or for conversion of consecutive to cyclic one-dimensional partitioning), encoded in binary code.

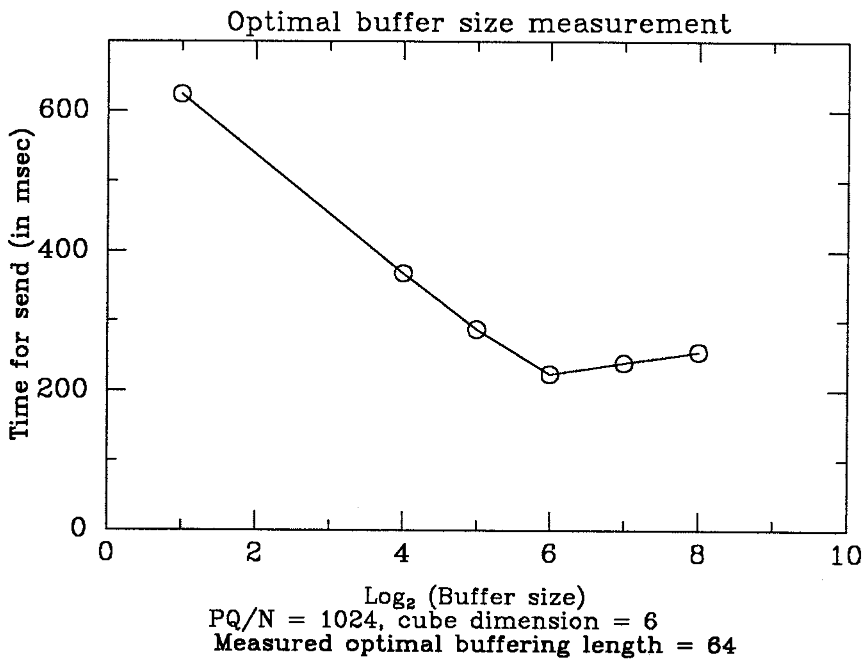


Figure 11: Performance measurements for optimum buffer size on the Intel iPSC.

to send blocks of length at least 64 floating-point numbers without buffering. Figure 11 illustrates the sensitivity of the performance to the choice of minimum unbuffered message size. Figure 12 shows the improvement in performance with optimum buffering compared to the unbuffered communication. Note that for sufficiently small cubes (or large data sets) the time required by the two schemes coincide.

On the iPSC, it is also possible to realize the *all-to-all personalized communication* by calling the iPSC router $2(N-1)$ times. However, the measured times of this are always inferior to that of the optimum buffering algorithm. The difference is from a factor of 5 to two orders of magnitude depending on the matrix size and cube size as observed in [14].

8.2 Two-Dimensional Partitioning

8.2.1 The Intel iPSC

We have implemented algorithm SPT as a step by step procedure. Pipelining is not possible. Moreover, on the Intel iPSC it is necessary to rearrange two-dimensional arrays into one-dimensional arrays before sending. Since the copy time is significant we arrive at an estimate for the time of a two-dimensional transpose of $T = (\frac{PQ}{N}t_c + \lceil \frac{PQ}{B_mN} \rceil \tau)n + 2\frac{PQ}{N}t_{copy}$. The growth rate is proportional to the number of matrix elements. There

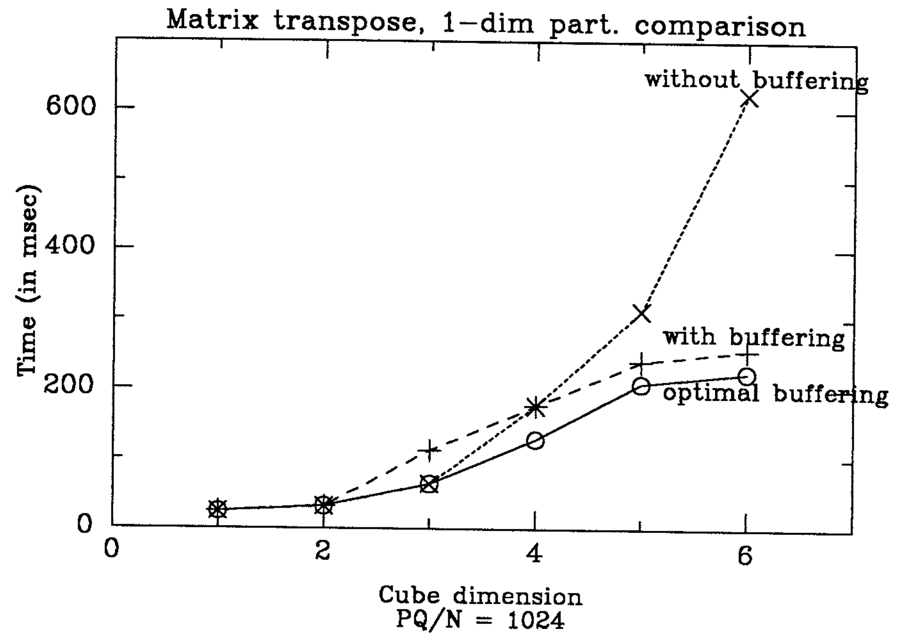


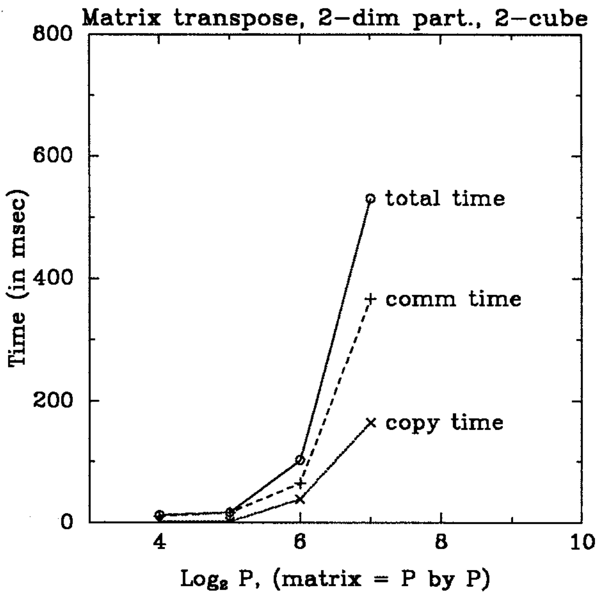
Figure 12: The effect of optimum buffering on performance for matrix transpose on the Intel iPSC.

is an exponential decay as well as a linear increase in the number of cube dimensions. Figure 13 shows measured values for the copy time, the communication time and the total time for a 2-cube and a 6-cube. As expected, the copy time for the 6-cube is lower than that for the 2-cube. Also, the communication time is essentially determined by the number of start-ups, which for the 6-cube remains the same for $PQ \leq 64$ KBytes.

Figure 14(a) shows the total transpose time as a function of the number of cube dimensions and matrix size. For small matrices the number of communication start-ups dominates and the total time increases with the number of cube dimensions, but as the matrix size increases the transpose time decreases with increased cube size.

On the Intel iPSC it is also possible to carry out the transpose operation by a direct send to the final destination. Figure 14(b) gives the times measured for matrix transpose using the routing logic alone. As the cube size increases the two-dimensional transpose algorithm yields a significantly better performance than the transpose time offered by the routing logic.

The time for matrix transposition with simultaneous conversion from Gray code to binary code conversion is shown in Figure 15. It is assumed that rows and columns have different encoding schemes. The Figure compares the $2n - 2$ steps naive algorithm and the n steps combined algorithm.



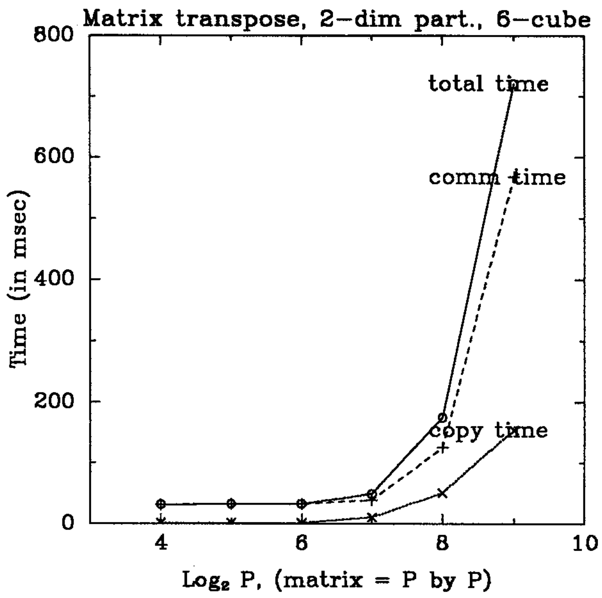
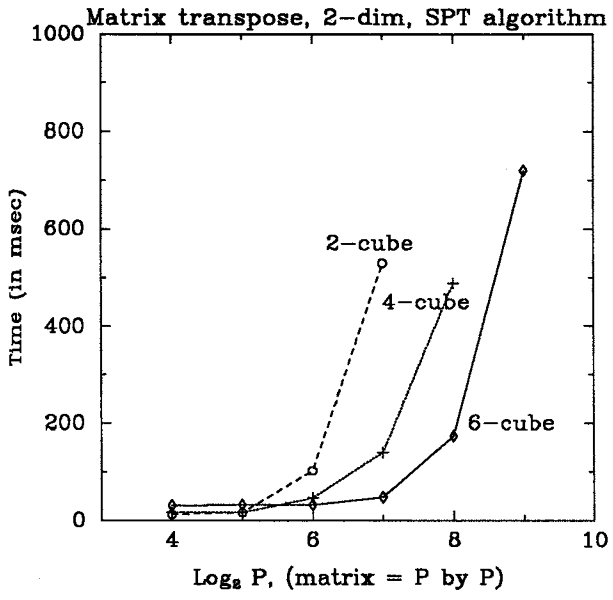


Figure 13: Performance measurements for a two-dimensional matrix transpose on the Intel iPSC.



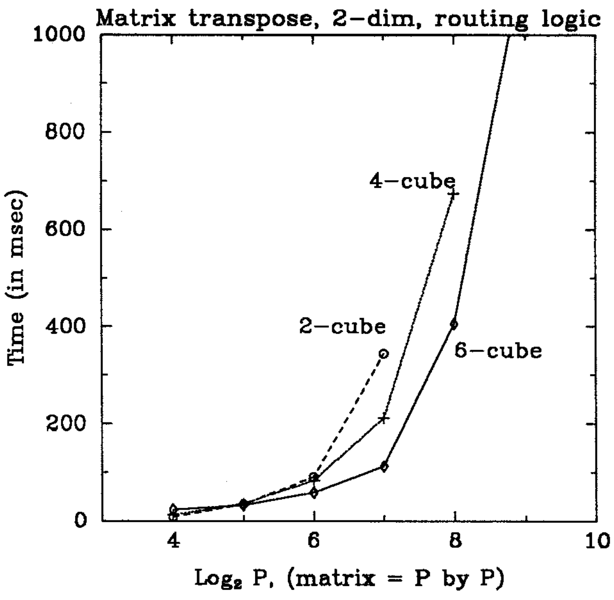


Figure 14: Measured times for a two-dimensional matrix transpose on the Intel iPSC using the SPT algorithm without pipelining (a) and using routing logic (b).

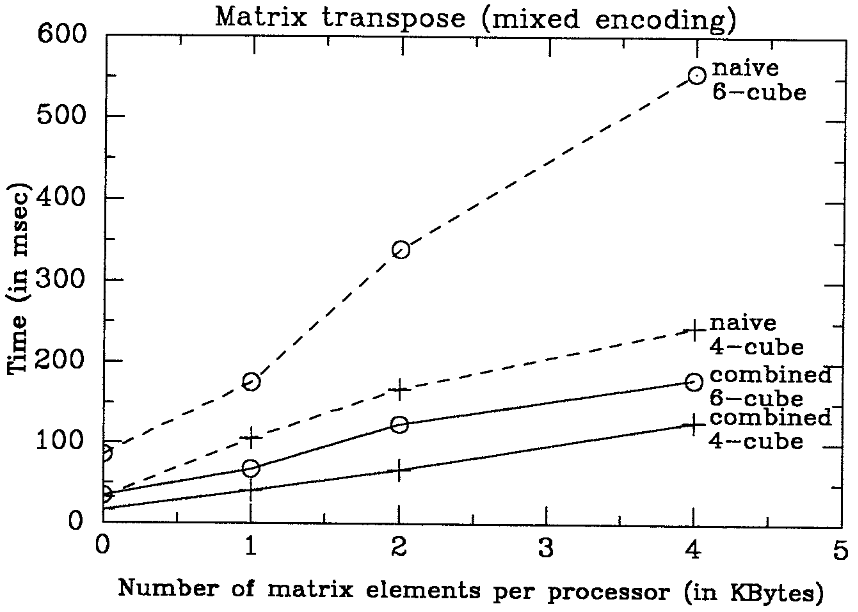


Figure 15: Measured times of transposing a matrix stored by mixed encoding of rows and columns by the naive and combined algorithms on the Intel iPSC.

8.2.2 The Connection Machine

We have also implemented the matrix transpose operation on the Connection Machine. It has a bit-serial, pipelined communication system. The recursive algorithm does not exploit this feature, but the routing logic does. Figure 16 shows the transpose time using the routing logic. Each processor holds one matrix element (32-bits). Figure 17 shows the transpose times for various number of matrix elements per processor, and for various number of processors. Figure 18 shows the transpose times for two fixed sized matrices on various sizes of the Connection Machine.

9 Comparison and Conclusion

It is of interest to compare the times for matrix transpose based on a one-dimensional partitioning and a two-dimensional partitioning. We now compare the complexity estimate for the two-dimensional transpose

$$T^{2d} = (\frac{PQ}{N}t_c + \lceil\frac{PQ}{B_m N}\rceil\tau)n + 2\frac{PQ}{N}t_{copy}$$

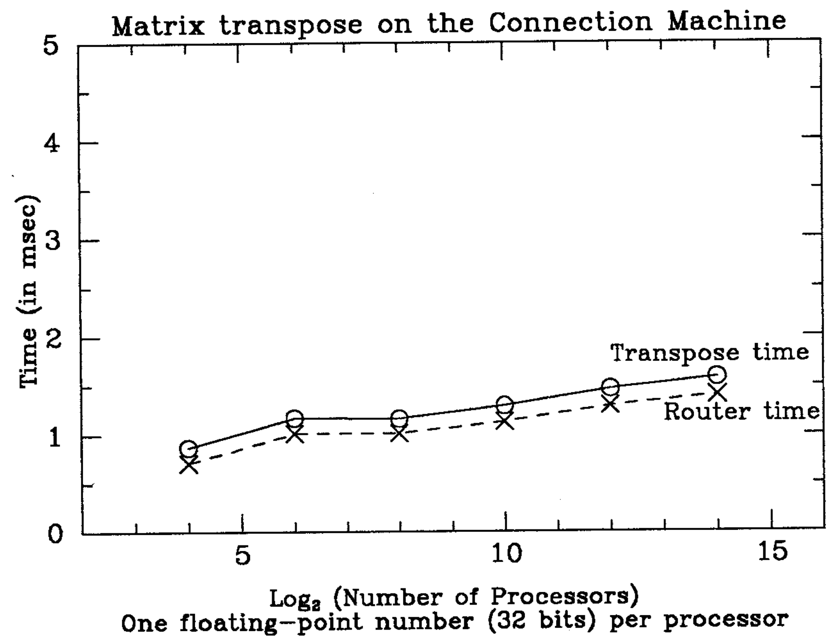


Figure 16: Matrix transpose on the Connection Machine. One element per processor.

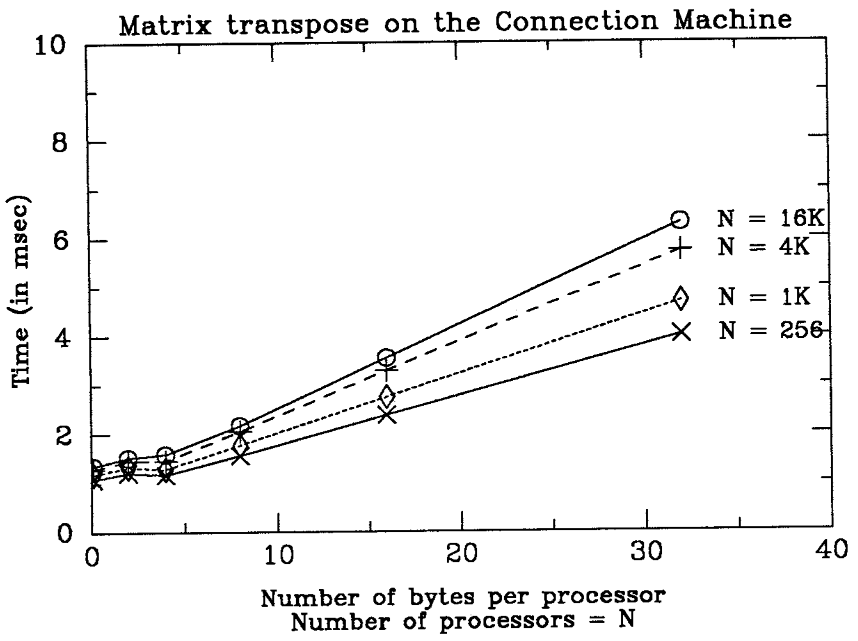


Figure 17: Matrix transpose on the Connection Machine. Multiple elements per processor.

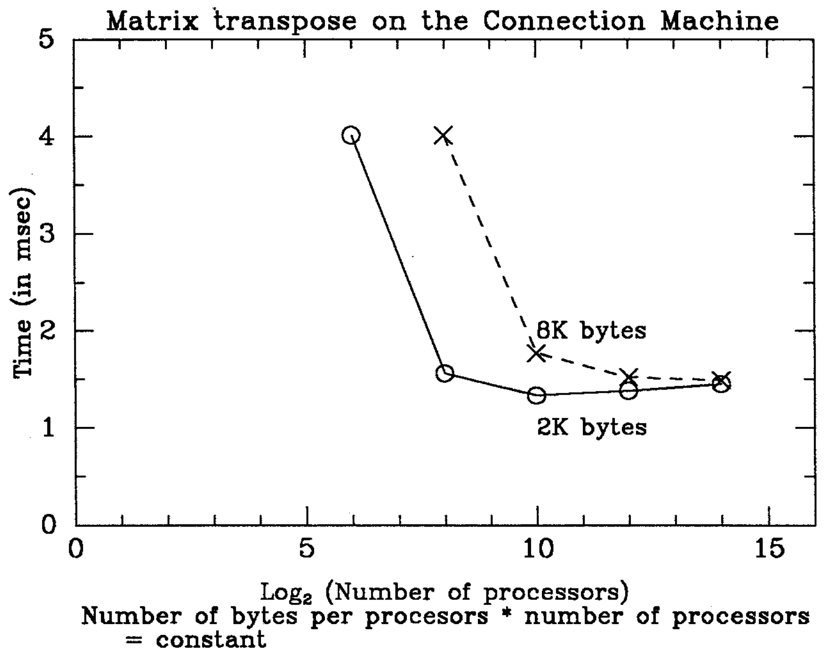


Figure 18: Matrix transpose on the Connection Machine as a function of the machine size.

with that for the one-dimensional transpose

$$\begin{aligned} T^{1d} &= (\min(n, \log\lceil\frac{PQ}{B_m N}\rceil)\lceil\frac{PQ}{2B_m N}\rceil + \min(N, \frac{PQ}{B_{copy} N}) - \min(N, \frac{PQ}{B_m N}) \\ &+ \lceil\frac{PQ}{2B_m N}\rceil \max(0, n - \log\lceil\frac{PQ}{B_{copy} N}\rceil))\tau \\ &+ n\frac{PQ}{2N}t_c + \frac{PQ}{2N}\max(0, n - \log\lceil\frac{PQ}{B_{copy} N}\rceil)t_{copy}. \end{aligned}$$

We have assumed that *one exchange* takes the same time as one send *or* one receive for *one-port* communication throughout the paper. With this model, the time for data transfers for the one-dimensional transpose is half of that of the two-dimensional transpose. If copy time is negligible, i.e., the time to copy B_m data is much less than a start-up time, then the number of start-ups for one-dimensional transpose is a factor of $\frac{1}{2}$ to 1 of that for the two-dimensional transpose. The factor $\frac{1}{2}$ applies for $\frac{PQ}{2N} \geq B_m$. By considering the copy time, we have two extreme cases. If $\frac{PQ}{N^2} \geq B_m$, the number of start-ups for the one-dimensional transpose is half of that for the two-dimensional transpose. If $\frac{PQ}{N} \leq B_{copy}$, it can be shown that the number of start-ups for the one-dimensional transpose is at most twice that for the two-dimensional transpose. In general, it can be shown that the number of start-ups for the one-dimensional transpose is a factor of $\frac{1}{2}$ to $\frac{B_m}{2B_{copy}} + \frac{1}{2}$ (which is 2.5 for the Intel iPSC) of that for the two-dimensional transpose.

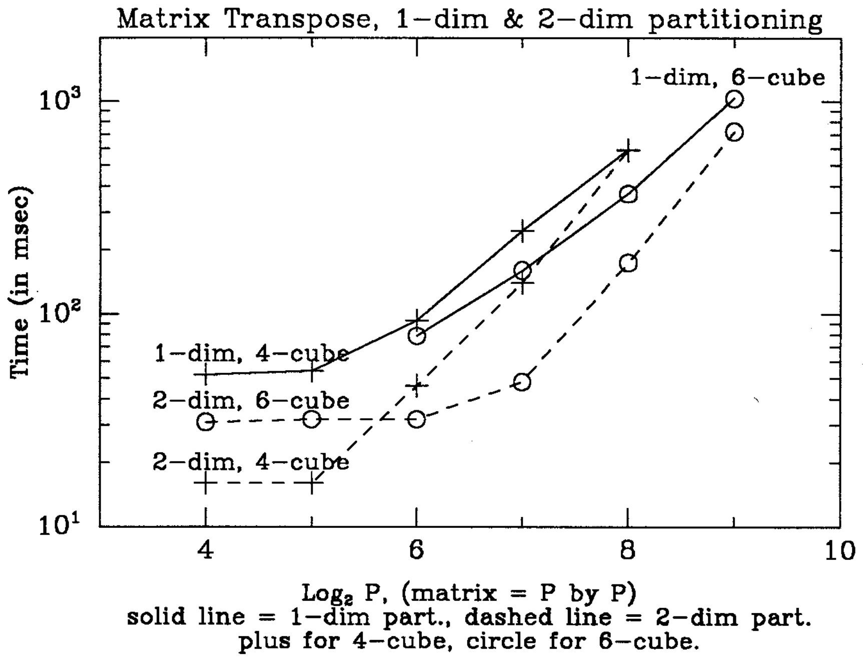


Figure 19: Comparison of the matrix transpose operation of one- and two-dimensional partitioned matrices on the Intel iPSC.

If the communication is restricted to one send *or* one receive at a time, the time for data transfers and the number of start-ups increase by a factor of two for the one-dimensional transpose. However, the complexity for the two-dimensional transpose remains the same. Therefore, the complexity of the two-dimensional transpose will be lower, or the same, as that of the one-dimensional transpose by a factor of 1 to $\frac{B_m}{B_{copy}}+1$.

Figure 19 gives the experimental result on the Intel iPSC.

With concurrent communication on multiple ports the transfer time for the two-dimensional partitioning decreases exponentially in the number of cube dimensions, but for the optimum packet size the number of start-ups is higher than for the one-dimensional partitioning. From the complexity estimates (one-dimensional partitioning)

$$T_{min}^{1d} = \frac{PQ}{2N}t_c + n\tau$$

and

$$T_{min}^{2d} = \begin{cases} (n+1)\tau + \frac{n+1}{2n}\frac{PQ}{N}t_c & \text{if } n \geq \sqrt{\frac{PQt_c}{N\tau}} \text{ approximately;} \\ (\frac{n}{2}+3)\tau + \frac{n+6}{2n+8}\frac{PQ}{N}t_c & \text{if } \sqrt{\frac{PQt_c}{2N\tau}} < n \leq \sqrt{\frac{PQt_c}{N\tau}} \text{ approximately and } \frac{n}{2} \text{ is even;} \\ (\frac{n}{2}+2)\tau + \frac{n+4}{2n+4}\frac{PQ}{N}t_c & \text{if } \sqrt{\frac{PQt_c}{2N\tau}} < n \leq \sqrt{\frac{PQt_c}{N\tau}} \text{ approximately and } \frac{n}{2} \text{ is odd;} \\ (\sqrt{\tau} + \sqrt{\frac{PQt_c}{2N}})^2 & \text{if } n \leq \sqrt{\frac{PQt_c}{2N\tau}}. \end{cases}$$

The optimum packet size is

$$B_{opt} = \begin{cases} \lceil \frac{PQ}{N(n+4)} \rceil & \text{for even } \frac{n}{2} \text{ and } n > \sqrt{\frac{PQt_c}{2N\tau}}; \\ \lceil \frac{PQ}{N(n+2)} \rceil & \text{for odd } \frac{n}{2} \text{ and } n > \sqrt{\frac{PQt_c}{2N\tau}}; \\ \sqrt{\frac{PQ\tau}{2Nt_c}} & \text{for } n \leq \sqrt{\frac{PQt_c}{2N\tau}}. \end{cases}$$

For $n \geq \sqrt{\frac{PQt_c}{N\tau}}$, the one-dimensional partitioning always yields a lower complexity than the two-dimensional partitioning. The difference is about one start-up time unless the cube is very small. For $\sqrt{\frac{PQt_c}{2N\tau}} < n \leq \sqrt{\frac{PQt_c}{N\tau}}$, the break even point (ignoring copy) can be computed to be

$$N \approx c\frac{r}{\log^2 r}$$

where $\frac{1}{2} < c < 1$ and $r = \frac{PQt_c}{\tau}$. For $n \leq \sqrt{\frac{PQt_c}{2N\tau}}$, the one-dimensional partitioning always yields a lower complexity than the two-dimensional partitioning.

In summary, if the copy time is ignored and communication is restricted to one port at a time, then the one-dimensional partitioning always yields a lower complexity than the two-dimensional partitioning. If the copy time is included then the two-dimensional partitioning yields a lower complexity for a sufficiently large cube. With concurrent communication on all ports the *Spanning Balanced n-Tree* (SBnT) routing can be used for the one-dimensional partitioning, and the copy times for one and two-dimensional partitioning should be comparable. The one-dimensional partitioning yields a lower complexity for a cube dimension n satisfying $n \geq \sqrt{\frac{PQt_c}{N\tau}}$ or $n \leq \sqrt{\frac{PQt_c}{2N\tau}}$.

In comparing the Intel iPSC with the Connection Machine we conclude that the latter performs a transpose about two orders of magnitude faster.

Acknowledgement

The authors express their appreciation for the comments made by one of the referees. The comments help improved the presentation of the results compared to the original version of this paper.

This work has in part been supported by the Office of Naval Research under Contract N00014-84-K-0043.

References

[1] J.O. Eklundh. A fast computer method for matrix transposing. *IEEE Trans. Computers*, C-21(7):801–803, 1972.

[2] Geoffrey C. Fox and Wojtek Furmanski. *Optimal Communication Algorithms on Hypercube.* Technical Report, California Institute of Technology, July 1986.

[3] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[4] Ching-Tien Ho and S. Lennart Johnsson. *Dimension Permutation on Boolean Cubes.* Technical Report , Dept. of Computer Science, Yale University, in preparation 1987.

[5] Ching-Tien Ho and S. Lennart Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *1986 Int. Conf. Parallel Processing*, pages 640–648, IEEE Computer Society, 1986. Tech. report YALEU/CSD/RR-483.

[6] Ching-Tien Ho and S. Lennart Johnsson. *Spanning Balanced Trees in Boolean cubes.* Technical Report YALEU/CSD/RR-508, Yale University, Dept. of Computer Science, January 1987.

[7] Ching-Tien Ho and S. Lennart Johnsson. *Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes.* Technical Report Report YALEU/CSD/RR-500, Yale University, Dept. of Computer Science, November 1986.

[8] S. Lennart Johnsson. Band matrix systems solvers on ensemble architectures. In *Algorithms, Architecture, and the Future of Scientific Computation*, University of Texas Press, 1985. (Report YALEU/CSD/RR-388, May 1985).

[9] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4(2):133–172, April 1987. (Report YALEU/CSD/RR-361, January 1985).

[10] S. Lennart Johnsson. *Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures.* Technical Report YALEU/CSD/RR-367, Yale University, Dept. of Computer Science, February 1985.

[11] S. Lennart Johnsson. *Odd-Even Cyclic Reduction on Ensemble Architectures and the Solution Tridiagonal Systems of Equations.* Technical Report YALE/CSD/RR-339, Department of Computer Science, Yale University, October 1984.

[12] S. Lennart Johnsson. Solving narrow banded systems on ensemble architectures. *ACM TOMS*, 11(3):271–288, November 1985. (Report YALEU/CSD/RR-418, November 1984).

[13] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Stat. Comp.*, 8(3):354–392, May 1987. (Report YALEU/CSD/RR-436, November 1985).

[14] S. Lennart Johnsson and Ching-Tien Ho. *Multiple tridiagonal systems, the Alternating Direction Method, and Boolean cube configured multiprocessors.* Technical Report YALEU/CSD/RR-532, Yale University, June 1987.

[15] Oliver A. McBryan and Eric F. Van de Velde. *Hypercube Algorithms and Implementations.* Technical Report , Courant Institute of Mathematical Sciences, New York University, November 1985.

[16] E M. Reingold, J Nievergelt, and N Deo. *Combinatorial Algorithms.* Prentice Hall, 1977.

[17] Yousef Saad and Martin H. Schultz. *Data Communication in Hypercubes.* Technical Report RR YALEU/DCS/RR-428, Dept. of Computer Science, Yale University, October 1985.

[18] Yousef Saad and Martin H. Schultz. *Topological properties of Hypercubes.* Technical Report RR YALEU/DCS/RR-389, Dept. of Computer Science, Yale University, June 1985.

[19] Harold S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Computers*, C-20:153–161, 1971.

[20] Quentin F. Stout and Bruce Wager. *Intensive Hypercube Communication I: Prearranged Communication in Link-Bound Machines.* Technical Report, Dept. of EECS, University of Michigan, June 1987.

[21] Quentin F. Stout and Bruce Wager. Passing messages in link-bound hypercubes. In *The 1986 Hypercube Conference*, SIAM, 1987.