Yale University
Department of Computer Science

Matrix Transposition on Boolean n-cube Configured Ensemble Architectures

Ching-Tien Ho and S. Lennart Johnsson

YALEU/DCS/TR-494
September 1986

This work by the Office Naval Research under Contracts No. N00014-85-K-0043 and N00014-82-K-0184 and in part by Thinking Machines Corporation.

Table of Contents

Matrix Transposition on Boolean n-cube Configured Ensemble Architectures

Ching-Tien Ho and S. Lennart Johnsson
Departments of Computer Science and
Electrical Engineering
Yale University
September 1986

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 that to a varying degree use the communication bandwidth of the 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 thereof. 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. Finally, experimental data are provided for the Intel iPSC and the Connection Machine.

1. Introduction

Matrix transposition is one of the basic operations frequently performed in linear algebra computations. Matrix transposition is useful in the solution of systems of linear equations by a variety of techniques. For instance, we have shown [7] that for the solution of tridiagonal systems of equations on Boolean n-cube configured architectures it may be beneficial with respect to performance to move all equations of a tridiagonal system to one processor and solve it locally instead of exploiting the maximum degree of concurrency using cyclic reduction with one equation per node. In the case of multiple tridiagonal systems the equations belonging to different systems are moved to distinct nodes. This data movement is equivalent to a matrix transpose operation. Multiple tridiagonal systems occur in the Alternating Direction Implicit (ADI) method and in the solution of Poisson's problem by the Fourier Analysis Cyclic Reduction (FACR) method.

In this paper we focus on the matrix transpose operation on Boolean n-cube architectures. The transpose can be formed recursively as described in for instance [10, 1, 5, 8]. Stone describes a mapping on to shuffle-exchange networks. Stone assumes that there is only one matrix element per node. We consider the case with multiple matrix elements per node and focus on the pipelining of communication operations as well as optimally using the communication bandwidth of the Boolean n-cube. In [5, 6] we described and analysed the complexity of a transpose algorithm for a two-dimensional mesh as well as a few algorithms for the transpose operation 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/d7 for the transpose of matrices partitioned one- or two-dimensionally, and for the Connection Machine [2] for two-dimensionally partitioned matrices.

The outline of the paper is as follows. In the next section we introduce the notation and

data structures used in this study. Then we present algorithms and carry out an analysis of the transpose operation for a one-dimensional partitioning of matrices, followed by a similar analysis of two-dimensional partitioning. We then describe some of the implementation issues that are somewhat particular for the actual machines used, but important for the interpretation of the experimental results which we present. A summary and conclusions follows.

2. Notation and Definition

In the case of an $N = 2^n$ processors Boolean n-cube and a $P \times Q$ matrix such that $P = 2^p$, $Q = 2^q$ and $p + q = n$, matrix elements can be assigned to distinct processors without any waste. One obvious assignment is to embed the matrix by encoding the row and column indices of matrix elements in binary code. Such an embedding does not preserve proximity. A *binary-reflected Gray code* [9] encoding of row and column indices preserves adjacency. This code is referred to as Gray code in the following and the encoding of i is $G(i)$. Depending on what other operations are being performed on the matrix data one or the other encoding may be preferable. The conversion from one kind of encoding to the other can be accomplished in $\log N - 1$ routing steps [5], where a pipelinable routing is given for the case where the processors can support communication on multiple ports (the routing paths can be made edge-disjoint). In the case of $P \times Q > N$ multiple matrix elements must be assigned to the same node in the Boolean cube. For $N < \max(P, Q)$ there is a choice between one-dimensional partitioning (strip) and two-dimensional partitioning (block).

For either kind of partitioning there is the additional choice of *cyclic* or *consecutive* assignment [5, 6] of matrix elements to partitions. In addition, there is the choice of binary or Gray code encoding. In [5, 6] we present algorithms and derive the communication complexities for both partitionings, element assignments, and encodings. Experimental data and some improved algorithms are presented here.

In a one-dimensional *cyclic* partitioning column (or row) j is assigned to partition $j \bmod N$ and in a one-dimensional *consecutive* partitioning column j is assigned to partition $\lfloor \frac{j}{\lceil \frac{Q}{N} \rceil} \rfloor$ with partitions labeled $0, 1, \ldots, N-1$. In a cube with $N = 2^n$ nodes the n lowest order bits of the binary encoded column (row) index determines the partition to which a column (row) is assigned in the cyclic partitioning, and analogously, the n highest order bits determines the partition assignment in the consecutive partitioning. The partitions are assigned to cube nodes through encoding in binary or Gray code. Figure 1 illustrates the two forms of one-dimensional partitioning.

In the two-dimensional partitioning we let N_r denote the number of partitions in the row direction and N_c the number of partitions in the column direction. The total number of partitions is $N_r \times N_c = N$. In the two-dimensional cyclic partitioning matrix element (i, j) is assigned to partition $(i \bmod N_r, j \bmod N_c)$. In the two-dimensional consecutive partitioning the matrix element (i, j) is assigned to partition $(\lfloor \frac{i}{\lceil \frac{P}{N_r} \rceil} \rfloor, \lfloor \frac{j}{\lceil \frac{Q}{N_c} \rceil} \rfloor)$. For a $P \times Q$ matrix partitioned by the consecutive strategy the highest order $\log N_r$ bits of the matrix row index determines the partition row index. Analogously, the $\log N_c$ highest order bits of the matrix column index determines the partition column index. We assume for simplicity that $N_r = 2^r$ and that $N_c = 2^c$. In cyclic storage, it is instead the last several bits of the matrix row and column indices that determines the assignment to a partition. The partitions can then be assigned to nodes in the cube by encoding the row and column indices of a partition in binary code or Gray code. Figure 2 illustrates the two forms of

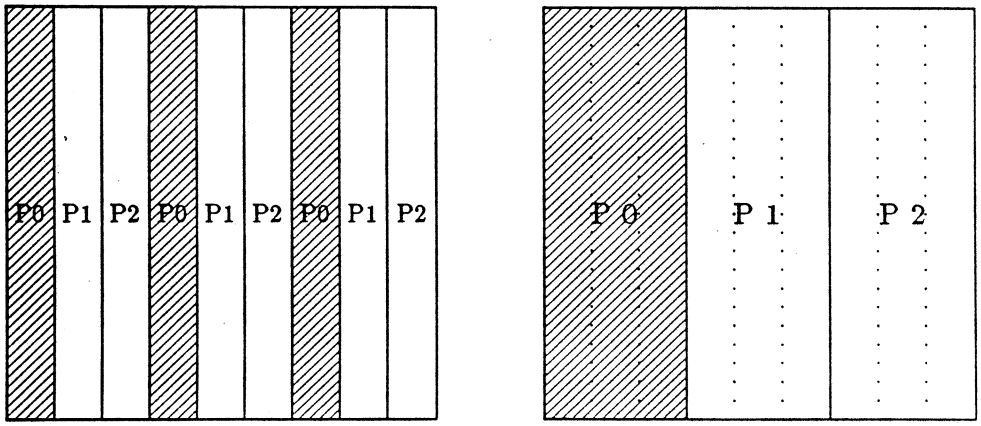

Figure 1: Cyclic and Consecutive one-dimensional partitioning.

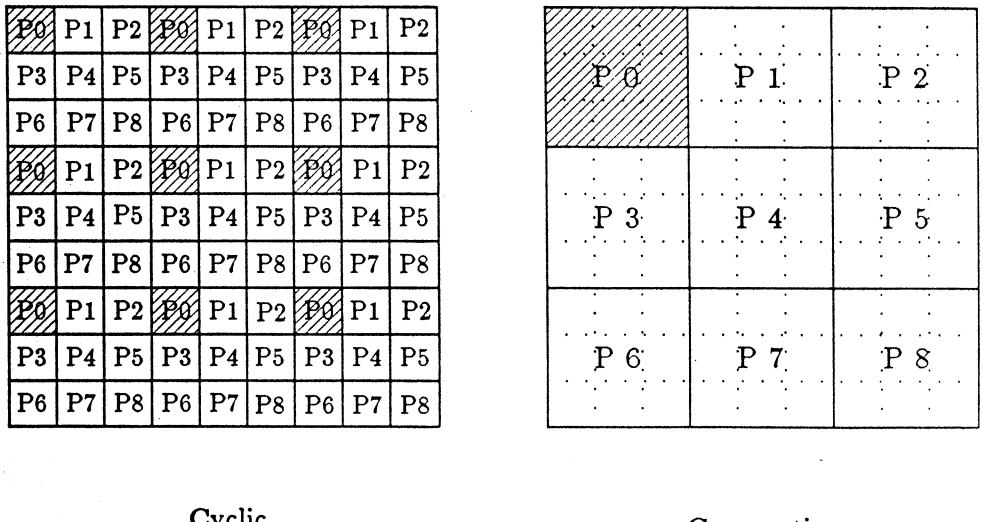

Figure 2: Cyclic and Consecutive two-dimensional partitioning.

two-dimensional partitioning.

We have now defined the different data structures and encodings that we consider. For the architecture we assume that it has packet oriented communication with a communications overhead, start-up time τ, and that the transmission time per element is t_c and a maximum packet size of B elements. The communication time is proportional to the number of communication links that is traversed. This model is applicable for the Intel iPSC. For a bit-serial machine, as the Connection Machine, pipelining can be used to make the overhead an additive factor. With the

current operating system for the Intel iPSC $\tau \approx 5ms$, $t_c \approx 1\mu sec/byte$ and $B_m = 1k$ bytes. For the algorithm description and analysis we consider two cases with respect to communication capabilities: communication restricted to one port at a time, and concurrent communication on all ports. The former is a good approximation of the capabilities of the Intel iPSC.

For the two-dimensional partitioning we assume that $n = r + c$ and write the processor address as $(ra_{r-1}ra_{r-2}\ldots ra_0ca_{c-1}ca_{c-2}\ldots ca_0)$ or $(ra||ca)$, where $ra = (ra_{r-1}ra_{r-2}...ra_0)$, $ca = (ca_{c-1}ca_{c-2}...ca_0)$ and '||' is the concatenation operator of two binary numbers.

3. One-dimensional Matrix Partitioning

3.1. Consecutive and Cyclic Storage

The transpose of a matrix from consecutive row to consecutive column partitioning has the same communication pattern as transpose from cyclic row to cyclic column partitioning. If $P, Q \geq N$ then each node needs to send data to all other nodes. For $P < N$ or $Q < N$ a subset of the nodes are either recipients or senders of data. Conversion between consecutive row and cyclic row (or column) partitioning also implies that all nodes send unique information to all other nodes, if $Q \geq N^2$ for column partitioning and $P \geq N^2$ for row partitioning. Consecutive partitioning is made on the $\log N$ highest order bits, cyclic partitioning on the $\log N$ lowest order bits of the row or column indices. Clearly, if less than $2 \log N$ bits are required for the encoding of row or column indices there is not sufficient data to require data to be sent from each node to every other node in the conversion. The amount of data that is sent from any node to any other node is $\frac{PQ}{N}$ in either of the two matrix transpose operations, or the cyclic/consecutive partitioning conversion. We formulate this observation as a proposition.

Proposition 3.1. *Conversion between any two of the following six embeddings all falls in the same class as all-to-all personalized communication [3], if $P \geq N^2$ for row partitioning or $Q \geq N^2$ for column partitioning, in that each processor sends the same amount*[1] *of personalized messages to all other processors.*[2]

1. *consecutive row storage.*
2. *consecutive column storage.*
3. *cyclic row storage.*
4. *cyclic column storage.*
5. *combination of consecutive row and cyclic row storage.*
6. *combination of consecutive column and cyclic column storage.*

For the transpose of a matrix partitioned consecutively by columns the partition assignment is changed from the $\log N$ highest order bits of the column index encoding to the $\log N$ highest order bits of the row index encoding. For cyclic partitioning it is instead the lowest order $\log N$ bits of the row and column indices that are of interest. For conversions between cyclic and consecutive partitioning only one index (row or column) is involved. It follows that if either the maximum row or column index requires less than $\log N$ bits for its encoding (or both), then the matrix

[1] or at most differs by 1 if P (or Q) is not multiple of N^2.

[2] Conversions between 1 and 5, 3 and 5, 2 and 6, 4 and 6, must have the encoded indices disjoint.

	Consecutive	Combined	Cyclic
Binary, Row	$(ra_{p-1}ra_{p-2}...ra_{p-n})$	$(ra_{p-i}ra_{p-i-1}...ra_{p-i-n+1})$	$(ra_{n-1}ra_{n-2}...ra_0)$
Binary, Column	$(ca_{q-1}ca_{q-2}...ca_{q-n})$	$(ca_{q-i}ca_{q-i-1}...ca_{q-i-n+1})$	$(ca_{n-1}ca_{n-2}...ca_0)$
Gray, Row	$(G(ra_{p-1}ra_{p-2}...ra_{p-n}))$	$(G(ra_{p-i}ra_{p-i-1}...ra_{p-i-n+1}))$	$(G(ra_{n-1}ra_{n-2}...ra_0))$
Gray, Column	$(G(ca_{q-1}ca_{q-2}...ca_{q-n}))$	$(G(ca_{q-i}ca_{q-i-1}...ca_{q-i-n+1}))$	$(G(ca_{n-1}ca_{n-2}...ca_0))$

Table 1: The processor address for matrix element $(ra_{p-1}ra_{p-2}...ra_0, ca_{q-1}ca_{q-2}...ca_0)$.

transpose operation corresponds to a few distinct single source-to-all (or sinks-from-all) personalized communications[3]. In the conversion case each processor communicates with a few others. With a relevant index field of at least $2 \log N$ bits the only difference for any two operations defined in Proposition 3.1 is the data structures internal to a node. Some of these issues will be discussed in the implementation section.

3.2. Binary and Gray Code Embeddings

The six partitionings in proposition 3.1 can be used in connection with either Gray code encoding or binary code encoding. There is a total of 12 kinds of matrix embeddings obtainable through the combination of one-dimensional partitioning strategies and encodings in the Boolean cube; namely, consecutive or cyclic partitioning or a combination of the two, row or column partitioning, binary or Gray code encoding. Most conversions between any two of the 12 kinds of storage methods are equivalent in terms of the global communication.[3] Table 1 shows the address of the processor to which the matrix element $(ra_{p-1}ra_{p-2}...ra_0, ca_{q-1}ca_{q-2}...ca_0)$ belongs for the 12 encodings.

3.3. Generic Algorithms

In the case that there are data elements for every processing node both before and after the data rearrangement, the communication is *all-to-all personalized communicaton.* In the case where only a few processing nodes contain data before or after the transformation it is of the form *many-to-all* or *all-to-many personalized communication.* In the extreme case it is of the type *one-to-all personalized communication.* These forms of communication are studied in detail in [3].

3.3.1. One-to-all personalized communication

The main result for *one-to-all personalized communication* is that such communication can be performed in lower bound time by routing according to a *Spanning Binomial Tree* (SBT) with communication restricted to one port at a time. Before the communication, only one processor, i.e., the source node, holds all PQ data elements. After the communication, every processor holds $\frac{PQ}{N}$ data elements. The communication time is $T_{min} = (1 - \frac{1}{N})PQt_c + \tau(\lceil \frac{PQ}{B_m} \rceil + \min(n, log\lceil \frac{B_m N}{PQ} \rceil) - 1)$, which is minimized for $B_m \geq \frac{PQ}{2}$. $T_{opt} = (1 - \frac{1}{N})PQt_c + n\tau$. The number of elements to be communicated from a node to each of the $N - 1$ other nodes is $\frac{PQ}{N}$.

With concurrent communication on all n ports routing according to a SBT is no longer optimal. The lower bound for concurrent communication is $T = \frac{1}{n}(1 - \frac{1}{N})PQt_c + n\tau$, but due to the fact that half of the nodes of a SBT are in one subtree the SBT routing requires a transmission time of at least $\frac{1}{2}PQt_c$. One nearly optimal (truly optimal for n prime) routing strategy is to use a *Balanced Spanning Tree* (BST). The routing time is $T \approx \sum_{i=1}^{n}(\frac{1}{n}\binom{n}{i}\frac{PQ}{N}t_c + \lceil \frac{1}{n}\binom{n}{i}\frac{PQ}{B_m N} \rceil \tau) = T \approx$

[8] except the conversions between binary code and Gray code encoding in which both use the same partitioning scheme in the same direction.

$\frac{1}{n}(1-\frac{1}{N})PQt_c + \sum_{i=1}^{n}(\lceil\frac{1}{n}\binom{n}{i}\frac{PQ}{B_mN}\rceil\tau$, which has a minimum of $T_{opt} = \frac{1}{n}(1-\frac{1}{N})PQt_c + n\tau$ for $B_m \geq \sqrt{\frac{2}{\pi}}\frac{PQ}{n^{3/2}}$. The speed-up of the transmission time of BST routing over SBT routing is a factor of $n/2$. Moreover, the maximum packet size is reduced approximately by a factor of n.

The BST routing divides the node set into approximately n equal subsets. An alternative routing for the case of concurrent communication on n ports, is to divide the data set ($\frac{PQ}{N}$) for each node into n equal parts (if $\frac{PQ}{N} \bmod n = 0$) and route the parts differently. For instance, the parts can be routed according to SBT's *rotated* with respect to each other, or a combination of ***rotation*** and ***reflection***. The minimum time for ***one-to-all personalized communication*** using n distinctly rotated spanning binomial trees is $T = \frac{1}{n}(1-\frac{1}{N})PQt_c + n\tau$, i.e., the same complexity as that of the lower bound. In a level-by-level algorithm the root has to send $\sum_{j=1}^{n}\binom{n-j}{i}\frac{1}{n}\frac{PQ}{N}$ elements over any port during step $n-i$. The lower bound is achieved for a maximum packet size $B_m \geq \max\binom{n}{i}\frac{1}{n}\frac{PQ}{N} \approx \sqrt{\frac{2}{\pi}}\frac{PQ}{n^{3/2}}$, $1 \leq i \leq n$.

For $\frac{PQ}{N} = k < n$ the BST routing has a lower time complexity for element transfers. For k SBT's the transfer time for optimally rotated spanning binomial trees is $(2^n-1)\frac{2^{\frac{n}{k}}-1}{2^n-1}\frac{PQ}{N}t_c$ 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 $\frac{PQ}{N} = k < n$.[4]

3.3.2. All-to-all personalized communication

For ***all-to-all personalized communication*** a simple exchange algorithm scanning through the dimensions of the cube attains the lower bound, $T_{min} = 2n(\frac{PQ}{2N}t_c + \tau)$, with communication restricted to one port at a time [3]. In each transfer $\frac{PQ}{2N}$ elements are transfered. The exchange algorithm effectively routes elements from a node to all other nodes according to a SBT. The spanning binomial trees rooted at different nodes are translated versions of each other. With concurrent communication on all n ports pipelining can be employed in the exchange algorithm, but the algorithm so modified is suboptimal. However, as in the case of a single source, routing based on balanced spanning trees attains the lower bound, $T_{min} = \frac{1}{2}(1-\frac{1}{N})\frac{PQ}{N}t_c + n\tau$, ignoring lower order terms [3]. The maximum number of elements transfered across an edge during any cycle is approximately $\frac{1}{n}(1-\frac{1}{N})\frac{PQ}{N}$. It is also possible to achieve the lower bound by employing rotated spanning binomial trees for $\frac{PQ}{N^2} \bmod n = 0$, and rotated/reflected SBT's otherwise.

The exchange algorithm [6] presented next performs the matrix transpose operation for one-dimensional partitionings embedded by binary encoding of the partition index. Assume that the matrix to be transposed 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 equally sized blocks. The transpose is formed by processor i exchanging its j^{th}

[4] If n is a multiple of k.

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 transpose is also two-dimensional, unless the number of columns is less than or equal to the number of processors. To complete the transpose operation after the interprocessor communication is completed, this two-dimensional data array can be transposed further locally, either explicitly or implicitly by indirect addressing.

An Exchange Algorithm

```
for j := n − 1 downto 0 do
    if (bit j of my-addr = 0) then
        exchange blocks ½N to N − 1 of my blocked array
            with my neighbor in dimension j
    else
        exchange blocks 0 to ½N − 1 of my blocked array
            with my neighbor in dimension j
    endif;
    shuffle my blocked array;
enddo
```

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.

A BST Algorithm

```
/* Let the format of msg be <source-addr, relative-addr, data>.  */
for all j ≠ myaddr do
    form msg for processor j = <myaddr, myaddr ⊕ j ⊕ 00..01_b0..0, data> and
        append to output-buf [b] where b is the base of myaddr ⊕ j.
loop n times
    send concurrently for all n output ports.
    receive concurrently for all n input ports.
    for each j do, 0 ≤ 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 ⊕ (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
        enddo
    enddo
endloop
```

For both the SBT and BST 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 locally, we can first perform a transformation locally such that block i is moved to block location $G(i)$, then carry out the above algorithms. The two operations can also be combined as described in the next section for two-dimensional partitioning.

4. Two-dimensional Partitioning

4.1. Relationships between different data structures and their encodings

As with one-dimensional partitioning there is a multitude of cases. If there is no particular reason for identifying a particular processor with a particular partition, then renaming of the processors suffices to realize the transpose. The processors that were assigned to column partitions will be assigned to row partitions after such a relabelling.

With the same number of partitions in the row and column directions, the transpose operation of a matrix partitioned by the consecutive strategy in both dimensions implies the same interprocessor communication as if the partitioning is made cyclicly in both dimensions (or by the same combination in both dimensions). An exchange of data takes place between distinct pairs of partitions. What matrix elements are exchanged depends on the partitioning strategy as is easily seen if the concatenated bitfield of the matrix row and column indices is partitioned with the first (or last) $\log N_r = \log N_c$ bits of the row and column halves of the index field making up the partition address. In [5, 6] we show that performing a matrix transpose on a matrix with partitions embedded in the cube by a binary code or Gray code encoding implies the same communication.

To transpose a matrix stored consecutively with respect to both row and columns to a form stored cyclically with respect to both row and columns, there exist a few alternatives, if the local data (submatrix) is stored row by row as a one dimensional array. We first state the complexity result as lemma.

Lemma 4.1. *A matrix stored consecutively (cyclicly) can be transposed and the storage form changed to cyclic (consecutive) in $2n$ communication steps for a $2^n \times 2^n$ matrix stored with one matrix element per node in an $2n$-cube.*

Let *exchange-row*(i) denote the sequence of exchange operations between rows as defined by the exchange algorithm described in pseudo code above with i being the block size for the first step. Hence, *exchange-row*(i) operates within each column subcube. Each subsequent step reduces the block size by a factor of 2 and doubles the number of blocks. In the exchange algorithm presented earlier, $i = PQ/2N$. *Exchange-column*(i) is similar. The optimal buffering technique can be applied in the exchange algorithm used for the consecutive/cyclic conversion. For the matrix transposition with different data structures we consider the following alternatives

1. Convert from consecutive-row partitioning to cyclic-row partitioning, i.e., *exchange-row* (Q/N_c); then convert from consecutive-column partitioning to cyclic-column partitioning, i.e., *exchange-column*(1); then (globally) transpose the matrix.
2. Transpose each partitioned matrix locally; then convert from consecutive-row partitioning, i.e., *exchange-row* (Q/N_c); then convert from consecutive-column partitioning to cyclic-column partitioning, i.e., *exchange-column*(1).

3. Convert from consecutive-column to cyclic-column partitioning, i.e., *exchange-column*(Q/N_c); then convert from consecutive-row to cyclic-row partitioning, i.e., *exchange-row*(1).

The first method requires $2\log N$ communication steps. The second method needs only $\log N$ steps. However a local matrix tranpose has to be performed first. The third method, while still requiring $\log N$ communication steps, does not require a local matrix transpose. It is eliminated by viewing the row as the column and vice versa when carrying out the consecutive/cyclic conversions. Note that the order between exchange-row and exchange-column can be reversed.

If the number of row and column partitions are different, then the transpose operation is no longer a pure exchange operation between a pair of processors. Some one-to-many and many-to-one communications are necessary. For example, assume that there are two row partitions and four column partitions. Then some partitions exchange data with one other partition, and some partitions with two other partitions. If *virtual* partitions are introduced in the row direction such that there are equally many row partitions, then the transpose operation becomes equivalent to the canonical case having as many row as column partitions. The number of virtual partitions is $2^{|\log N_r - \log N_c|}$. With virtual partitions one dimension is "collapsed" to a certain degree.

Conversion between cyclic storage and consecutive storage 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 $P \geq N_r^2$. In the two-dimensional conversion the assignment of matrix rows to partitions is changed from being determined by the last $\log N_r$ bits of the matrix row index to the first $\log N_r$ bits, or vice versa, and the column assignment is changed similarly according to the last and first $\log N_c$ bits of the matrix column index. The source cube node address is defined by the concatenated last $\log N_r$ bits of the row index and $\log N_c$ bits of the column index, and the destination address by the concatenated first $\log N_r$ bits of the matrix row index and $\log N_c$ bits of the column index. Clearly, by a suitable permutation of the bits in the concatenated row and column index encoding the two-dimensional conversion is equivalent to a one-dimensional conversion on a cube with $\log N_r + \log N_c = \log N$ dimensions.

4.2. Algorithms

We consider the transposition operation for binary encoding first. Define $tr(i)$ to be the function which maps the address of partition $i = (ra||ca)$ to the address of the transposed partition, i.e., $tr(i) = (ca||ra)$. Let $D(i) = |ra \oplus ca|$, i.e., the value of $D(i)$ is equal to the number of bits that differ in ra and ca. The distance between i and $tr(i)$ is $2D(i)$. We assume that there are equally many row and column partitions.

The *Single path Recursive Transpose(SRT)* algorithm [6] uses one path from node i to $tr(i)$. Paths for different i are edge-disjoint, and pipelining of communications can be employed to reduce the communication complexity. The *Dual paths Recursive Transpose(DRT)* algorithm is a straightforward improvment of the *SRT* algorithm in that two directed edge-disjoint paths are established from each source node to its corresponding destination node. In the *Multiple paths Recursive Transpose(MRT)* algorithm, we partition all the nodes into sets having equivalent properties with respect to an operator (defined later). We show that the paths of any two nodes belonging to different sets are edge-disjoint. We then prove that all the nodes in the same set share the same set of edges, but use them during different cycles.

4.3. The Single path Recursive Transpose(SRT) Algorithm

The *Single path Recursive Transpose(SRT)* algorithm [6] for a two-dimensional consecutively partitioned matrix exchanges data between the upper right $P/2 \times Q/2$ submatrix ($ra_{\frac{n}{2}-1} = 0$, $ca_{\frac{n}{2}-1} = 1$) and the lower left submatrix ($ra_{\frac{n}{2}-1} = 1$, $ca_{\frac{n}{2}-1} = 0$) in two steps. The transpose operation is completed by recursively applying the operation to each of the four submatrices. The implied routing corresponds to directed edge-disjoint paths from each node i to $tr(i)$. For each source-destination pair there is a single path. This path only goes through the appropriate dimensions of the cube corresponding to the bits of the source node address i that need to be complemented to become the destination node address $tr(i)$. The routing order for the dimensions that need to be routed is the same for all nodes, for instance highest to lowest order for both row and column encoding, i.e., $ra_{\frac{n}{2}-1}, ca_{\frac{n}{2}-1}, ra_{\frac{n}{2}-2}, ca_{\frac{n}{2}-2}, \ldots, ra_0, ca_0$. The length of the path of node i is $2D(i)$. 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 $\lceil \frac{PQ}{B_m N} \rceil + n - 1$. The nodes which are not on the anti-diagonal can either finish the transposition early 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 $(\frac{PQ}{B_m N} + n - 1)(B_m t_c + \tau)$. The optimal packet size, B_{opt}, is $\sqrt{\frac{PQ\tau}{N(n-1)t_c}}$ and the minimum time, $T_{min} = (\sqrt{\frac{PQ}{N}t_c} + \sqrt{(n-1)\tau})^2$.

4.4. The Dual paths Recursive Transpose(DRT) Algorithm

The *SRT* algorithm can be improved by establishing two directed edge-disjoint paths between i and $tr(i)$ for all i. In addition to the paths used in the *SRT* algorithm, a second path is defined by permuting row and column dimensions pairwise to yield a routing order selected from $ca_{\frac{n}{2}-1}, ra_{\frac{n}{2}-1}, ca_{\frac{n}{2}-2}, ra_{\frac{n}{2}-2}, \ldots, ca_0, ra_0$. The two directed paths for a particular i are edge-disjoint (as observed in [4] for the solution of tridiagonal systems on Boolean cubes). Moreover, the two directed paths for any i are edge-disjoint with respect to all paths for other i. 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 *SRT* algorithm. The communication complexity is $(\lceil \frac{PQ}{2B_m N} \rceil + n - 1)(B_m t_c + \tau)$, 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 *SRT* algorithm it suffices that each node supports a total of n concurrent send or receive operations, whereas for the *DRT* algorithm n send operations concurrently with n receive operations are required for each node. Uni-directional communication suffices for the *SRT* algorithm, but bidirectional communication is required for the *DRT* algorithm.

4.5. The Multiple paths Recursive Transpose(MRT) Algorithm

For the *Multiple paths Recursive Transpose(MRT)* algorithm we define $2D(i)$ paths, labeled $0, 1, \ldots, 2D(i)-1$, between nodes i and $tr(i)$. The paths differ in the order in which the dimensions are routed. All paths have the same length. Let $\alpha_{D(i)-1}, \alpha_{D(i)-2}, \ldots, \alpha_0, \beta_{D(i)-1}, \beta_{D(i)-2}, \ldots, \beta_0$ be the sequence of dimensions that need to be routed in descending order. We describe a pair of paths as a sequence of dimensions.

$$path\ p = \begin{cases} \alpha_{(p+D(i)-1) \bmod D(i)}, \beta_{(p+D(i)-1) \bmod D(i)}, \alpha_{(p+D(i)-2) \bmod D(i)}, \beta_{(p+D(i)-2) \bmod D(i)}, \cdots, \alpha_p, \beta_p. \\ \quad 0 \le p < D(i); \\ \beta_{(j+D(i)-1) \bmod D(i)}, \alpha_{(j+D(i)-1) \bmod D(i)}, \beta_{(j+D(i)-2) \bmod D(i)}, \alpha_{(j+D(i)-2) \bmod D(i)}, \cdots, \beta_j, \alpha_j. \\ \quad j = p - D(i), D(i) \le p < 2D(i). \end{cases}$$

For example, if $i = (1001||0100)$, then $ra = 1001, ca = 0100, D(i) = 3$ and $tr(i) = (ca||ra) = (0100||1001)$. The distance between i and $tr(i)$ is 6. The 6 paths are defined as follows.

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

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 $(p-1) \bmod D(i)$, if $0 \le p < D(i)$. For $D(i) \le p < 2D(i)$, path p can be derived by a right rotation of two steps of path $((p-1) \bmod D(i)) + D(i)$ and also by permuting row and column dimensions pairwise of path $(p-1) \bmod D(i)$. Note that path 0 is the same as the path defined in the *SRT* algorithm. Paths 0 and $D(i)$ are the two paths defined for node i in the *DRT* algorithm.

Definition 4.1. Let i, j be two nodes with $i = (ra'||ca')$ and $j = (ra''||ca'')$. Define a relationship $\sim_{ad}$ between i and j such that $i \sim_{ad} j$ iff $ra' + ca' = ra'' + ca''$, i.e., i and j are on the same anti-diagonal. Note that if $i \sim_{ad} j$ and $j \sim_{ad} k$ then $i \sim_{ad} k$.

Definition 4.2. Define $edge(i,p,e)$ to be the function which returns the e^{th} directed edge of path p of node i (starting 1^{st} edge). We also define *Edges*, *OddEdges*, *EvenEdges* and *Paths* as follows.

$$Edges(i,e) = \{edge(i,p,e) | \forall\ 0 \le p < 2D(i)\}.$$

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

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

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

Definition 4.3. Define $Nodes(i,e)$ to be the function which returns the set of nodes upon which the directed edges in $Edges(i,e)$ terminate. Define $OddNodes(i)$ and $EvenNodes(i)$ to be the set of nodes on which the set of directed edges $OddEdges(i)$ and $EvenEdges(i)$ terminate, respectively.

$$OddNodes(i) = \bigcup_{\forall \text{ odd } e} Nodes(i,e)$$

$$EvenNodes(i) = \bigcup_{\forall \text{ even } e} Nodes(i,e)$$

Definition 4.4. Let i, j be two nodes. Define a relationship operator $\sim_s$ such that $i \sim_s j$ iff $i \sim_{ad} j$ and $i \oplus tr(i) = j \oplus tr(j)$. If $i \sim_s j$ and $j \sim_s k$ then $i \sim_s k$.

Note that $i \oplus tr(i) = j \oplus tr(j)$ implies $D(i) = D(j)$, but $D(i) = D(j)$ does not imply $i \oplus tr(i) = j \oplus tr(j)$. There exists i, j such that $i \sim_{ad} j$ and $i \oplus tr(i) \neq j \oplus tr(j)$. Also there exists i, j such that $i \not\sim_{ad} j$ and $i \oplus tr(i) = j \oplus tr(j)$.

Definition 4.5. The set of paths defined for a set of nodes is said to be (u,n)-disjoint if each node in the set can send out a packet of fixed length through all the paths originating from it during cycles $1+i*n, 2+i*n, \ldots, u+i*n, \forall\, i \geq 0$, without routing conflicts, i.e., messages originating from different nodes will not be routed over the same edge during the same cycle.[5]

To describe the algorithm we first prove the following properties.

1. Path p_1 and p_2 of node i are edge-disjoint, $\forall\ 0 \leq p_1, p_2 < 2D(i), p_1 \neq p_2$.
2. If $i \not\sim_s j$ then $Paths(i) \cap Paths(j) = \phi$.
3. The set of all paths for the nodes in the set induced by the relationship $\sim_s$ is $(2, 2D(i))$-disjoint.

Lemma 4.2. *Path p_1 and p_2 of node i are edge-disjoint, $\forall\ 0 \leq p_1, p_2 < 2D(i), p_1 \neq p_2$.*

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 4.3. *If $D(i) > 0$, then the set of nodes $OddNodes(i)$ and $EvenNodes(i)$ have the following properties*

- $i \not\sim_{ad} j$, $D(j) = D(i) - 1, \forall\, j \in OddNodes(i)$,
- $i \sim_{ad} j$, $i \oplus tr(i) = j \oplus tr(j)$ *(which implies $D(j) = D(i)$)*, $\forall\, j \in EvenNodes(i)$.

Proof.

In traversing an edge in $OddEdges(i)$, we complement one of the $D(i)$ bits of the $\frac{n}{2}$ high (low) order bits which differ from the corresponding low (high) order bit. In traversing an edge in $EvenEdges(i)$, we complement the low (high) order bit of the corresponding high (low) order bit that was complemented in traversing the preceding odd edge. Let $v_1 = Nodes(i, 2h) = (ra||ca)$, $v_2 = Nodes(i, 2h+1) = (ra'||ca')$ and $v_3 = Nodes(i, 2h+2) = (ra''||ca'')$, $0 \leq h < D(i)$. From the definition of paths either $ra' = ra + 2^x, ca' = ca$ or $ra' = ra, ca' = ca - 2^x$ for some x satisfying $ra_x = 0$, $ca_x = 1$; or $ra' = ra - 2^x, ca' = ca$ or $ra' = ra, ca' = ca + 2^x$ for some x satisfying $ra_x = 1, ca_x = 0$. These conditions imply $ra + ca \neq ra' + ca'$,i.e., $v_1 \not\sim_{ad} v_2$, and $|ra'||ca'| = |ra||ca| - 1$, i.e., $D(v_2) = D(v_1) - 1$. Furthermore, $ra'' = ra + 2^x, ca'' = ca - 2^x$ for some x satisfying $ra_x = 0, ca_x = 1$ or $ra'' = ra - 2^x, ca'' = ca + 2^x$, for some x satisfying $ra_x = 1, ca_x = 0$. Hence, $ra + ca = ra'' + ca''$, i.e., $v_1 \sim_{ad} v_3$. Also, $ra \oplus ca = ra'' \oplus ca''$, i.e., $(ra||ca) \oplus (ca||ra) = (ra''||ca'') \oplus (ca''||ra'')$ which implies $v_1 \oplus tr(v_1) = v_3 \oplus tr(v_3)$.

∎

Corollary 4.1. $i \sim_s l, \forall\, l \in EvenNodes(i)$.

Lemma 4.4. *If $i \not\sim_{ad} j$, then $Paths(i) \cap Paths(j) = \phi$.*

Proof. It is sufficient to prove $Paths(i) \cap Paths(j) = \phi$ by proving $EvenNodes(i) \cap EvenNodes(j) = \phi$ and $EvenNodes(i) \cap OddNodes(j) = \phi$. From lemma 4.3, $EvenNodes(i) \sim_{ad} i$, $EvenNodes(j) \sim_{ad} j$. Since $i \not\sim_{ad} j$, we have $EvenNodes(i) \not\sim_{ad} EvenNodes(j)$, which implies $EvenNode(i) \cap EvenNode(j) = \phi$.

[5] Note that the (u,n)-disjoint definition does not imply that the paths from the different source nodes are edge-disjoint.

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

1. If $D(i) = D(j)$, then by lemma 4.3 $D(v_1) = D(v_2) + 1$ where $v_1 \in EvenNodes(i)$, $v_2 \in OddNodes(j)$. So, $EvenNodes(i) \cap OddNodes(j) = \phi$.
2. If $D(i) > D(j)$, then $D(v_1) > D(v_2)$ where $v_1 \in EvenNodes(i)$, $v_2 \in Oddnodes(j)$. So, $EvenNodes(i) \cap OddNodes(j) = \phi$.
3. If $D(i) < D(j)$, we show $EvenNodes(j) \cap OddNodes(i) = \phi$ instead by a similar argument as in case 2.

∎

Lemma 4.5. *If $i \sim_{ad} j$ and $i \not\sim_s j$, then $Paths(i) \cap Paths(j) = \phi$.*

Proof. Assume $EvenNodes(i) \cap EvenNodes(j) = \phi$, then there exists one node v such that $v \in EvenNodes(i)$ and $v \in EvenNodes(j)$. By corollary 4.1, $v \sim_s i, v \sim_s j$, i.e., $i \sim_s j$ which is a contradiction. So, $EvenNodes(i) \cap EvenNodes(j) = \phi$. Also by lemma 4.3, $EvenNodes(i) \cap OddNodes(j) = \phi$. Hence, $Paths(i) \cap Paths(j) = \phi$.

∎

Lemma 4.6. *If $i \not\sim_s j$ then $Paths(i) \cap Paths(j) = \phi$.*

Proof. It follows from lemmas 4.4 and 4.5.

∎

Lemma 4.7. *The set of paths defined for the nodes in the same set induced by the relationship $\sim_s$ is $(2, 2D(i))$-disjoint.*

Proof. We first prove that the paths of the nodes defined by the $\sim_s$ relationship are $(1, 2D(i))$-disjoint. The proof is by induction on the routing cycles. During cycles 1 and 2, the routed edges are clearly disjoint by Lemma 4.3. Assume that during cycles $2n-1$ and $2n$, $n > 0$, the routing is also edge-disjoint. If $n = D(i)$, then all the routing is complete. During the next two cycles the routing is restarted and there is no edge conflict. If $n \neq D(i)$, then consider the $2D(i)$ edges directed into some node l at distance $2n$ from i as well as the $2D(i)$ edges directed out from node l. Let $\alpha_{D(i)-1}, \alpha_{D(i)-2}, ..., \alpha_0, \beta_{D(i)-1}, \beta_{D(i)-2}, ..., \beta_0$ be the corresponding $2D(i)$ dimensions in descending order. If an edge used during cycle $2n-1$ is in dimension α_x (i.e., the edge used during cycle $2n$ is in dimension β_x) then the edges used during the following two cycles are in dimensions $\alpha_{(x-1) \bmod D(i)}$ and $\beta_{(x-1) \bmod D(i)}$ respectively. If the edge used during cycle $2n-1$ is in dimension β_x then the edges used during the following two cycles are in dimensions $\beta_{(x-1) \bmod D(i)}$ and $\alpha_{(x-1) \bmod D(i)}$ respectively. Hence, the edges used during the following two cycles are all distinct and it follows that the paths are $(1, 2D(i))$ disjoint.

To show that the paths are $(2, 2D(i))$-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 l 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 v_1 to node v_2 and even edges directed from node v_3 to node v_4 where $i \sim_s v_1 \sim_s v_4$, $i \not\sim_s v_2$ and $i \not\sim_s v_3$.

∎

For the routing, the data from node i is split into $4D(i)$ packets of size $\lceil \frac{PQ}{4ND(i)} \rceil$ each. The packets are sent during the first two cycles. The first $2D(i)$ packets will arrive at the destination node, $tr(i)$, after $2D(i)$ 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 $D(i)$ for $1 \leq D(i) \leq \sqrt{\frac{PQt_c}{8N\tau}}$ and increases for $\sqrt{\frac{PQt_c}{8N\tau}} \leq D(i)$. The transpose time for $D(i) = 1$ and $D(i) = \frac{PQt_c}{8N\tau}$ 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 $\lfloor \frac{n}{2D(i)} \rfloor * 4D(i)$ packets. In fact, the data sent from node i can be split into $4kD(i)$ packets instead of $4D(i)$ packets. The whole routing completes in $2kD(i)+1$ cycles. Hence, $T = (2kD(i)+1)(\tau + \frac{PQ}{4kD(i)N}), 1 \leq D(i) \leq \frac{n}{2}$. The optimal k is $\sqrt{\frac{PQt_c}{2N\tau}}\frac{1}{2D(i)}$ and $T_{min} = (\sqrt{\tau} + \sqrt{\frac{PQt_c}{2N}})^2$. Notice that T_{min} is valid only when $k \geq 1$, which implies $\sqrt{\frac{PQt_c}{2N\tau}} \geq n$.

Theorem 4.1. *The total matrix transpose time by the MRT algorithm is*

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

and the maximal packet size is

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

Proposition 4.1. *The matrix transposition time is at least* $n\tau + \frac{PQ}{2N}t_c$.

Proof. The minimum number of start-ups is determined by the largest 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 $\sqrt{N}/2 \times \sqrt{N}/2$ submatrix. There are $N/4$ nodes. Each node has to send $\frac{PQ}{N}$ data to some node outside the submatrix. There are two dimensions per node that connects to nodes outside of the submatrix, i.e., a total of $2N/4$ 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 P by Q matrix stored in $N = PQ$ (P by Q) processors by binary

encoding of row and column indices, matrix element (i,j) is stored in processor $i||j$ and matrix element (j,i) is stored in processor $j||i$. The two dimensional transpose algorithms described above can be viewed as a permutation between processor $ra||ca$ and processor $ca||ra$, $\forall 0 \leq ra < P, ca < Q$. For Gray code encoding of row and column indices, matrix element (i,j) is stored in processor $G(i)||G(j)$ and matrix element (j,i) is stored in processor $G(j)||G(i)$. 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.

4.6. 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 [6], 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.[6] The total number of routing steps is $2n - 2$. However, the number of routing steps can be reduced to n by combining the transpose[7] and the conversion operations. Pipelining still can be applied. For simplicity, we describe the non-pipelined version. Similar to the *SRT* algorithm, the combined algorithm is composed of $n/2$ iterations. Each iteration contains two routing steps. In iteration i, $0 \leq i < n/2$, bit $\frac{n}{2} - i - 1$ of the row and column indices is changed by sending data through the corresponding dimensions. With the rows encoded in binary code and the columns in Gray code, matrix block (i,j) is stored in processor $i||G(j)$ and matrix block (j,i) is stored in processor $j||G(i)$. The direct transpose permutation is defined by exchanging data between processor $r||c$ and processor $G^{-1}(c)||G(r)$.

During the first iteration, the upper right block $(0xx..x||1xx..x)$ and the lower left block $(1xx..x||0xx..x)$ are exchanged in two steps. Neither the row or column conversions for the two encodings affect 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 3.

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[6]). Figure 4 shows the four iterations with $n = 8$, in which c means *clockwise rotation* and cc means *counterclockwise rotation*. The algorithm is presented below.

```
/* Note:  my-addr is of the form (ra||ca).  */
/* The second argument of 'send' and 'recv' represent the cube dimension */
```

[6] The two conversions can also be performed after the transpose.
[7] i.e., the *SRT* algorithm

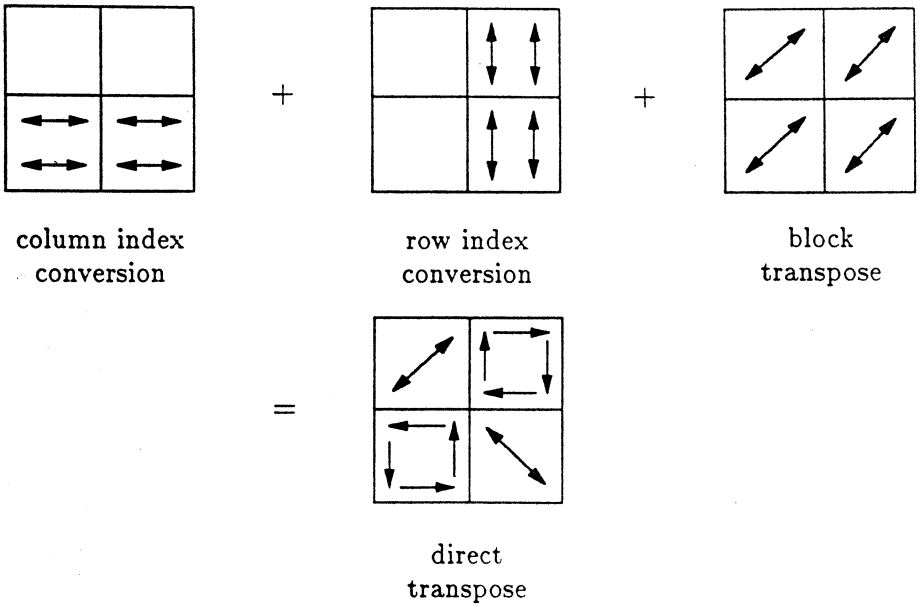

Figure 3: Transpose of a matrix stored by binary code encoding of row index and Gray code encoding of column index.

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

Figure 5 shows the measured time to transpose a matrix by mixed encoding of rows and columns by using the $2n-2$ steps naive algorithm and the n steps combined algorithm on the Intel iPSC.

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

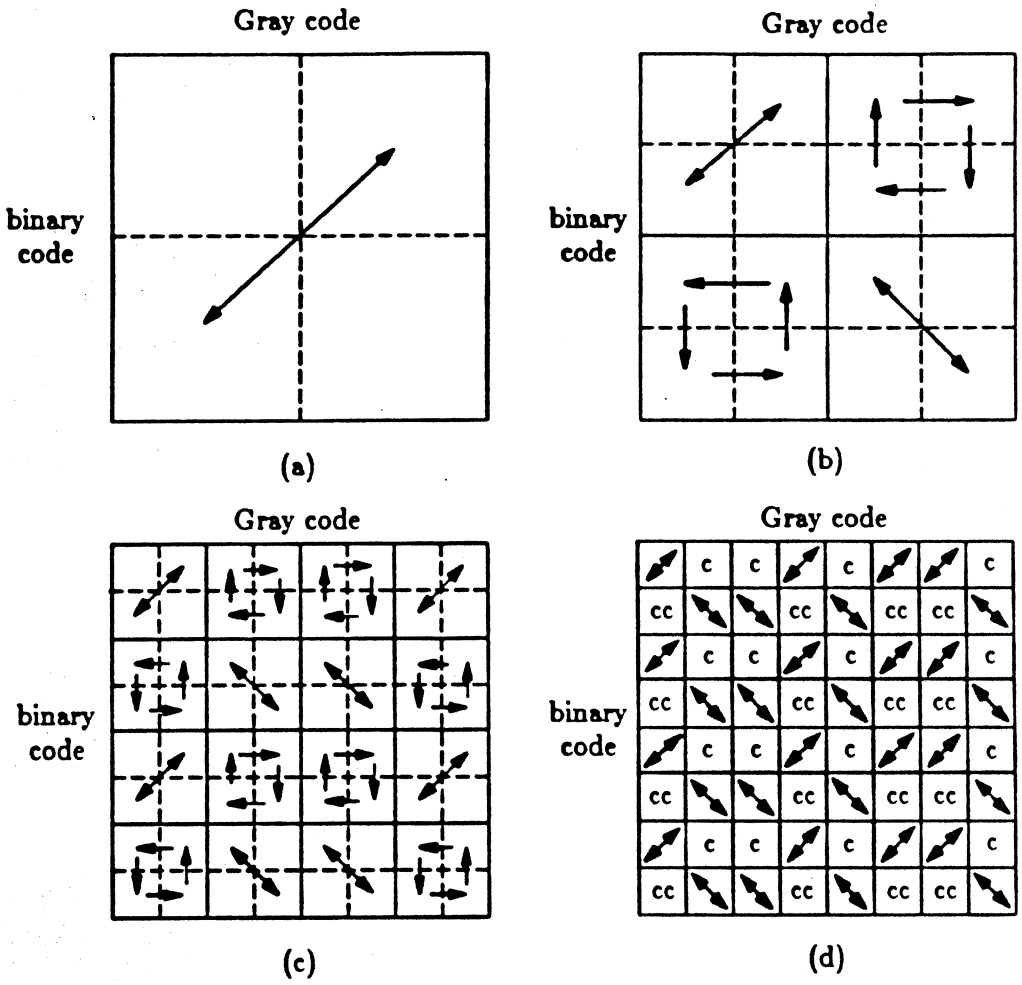

Figure 4: Transpose of a matrix stored by mixed encoding of rows and columns in an 8-cube.

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.

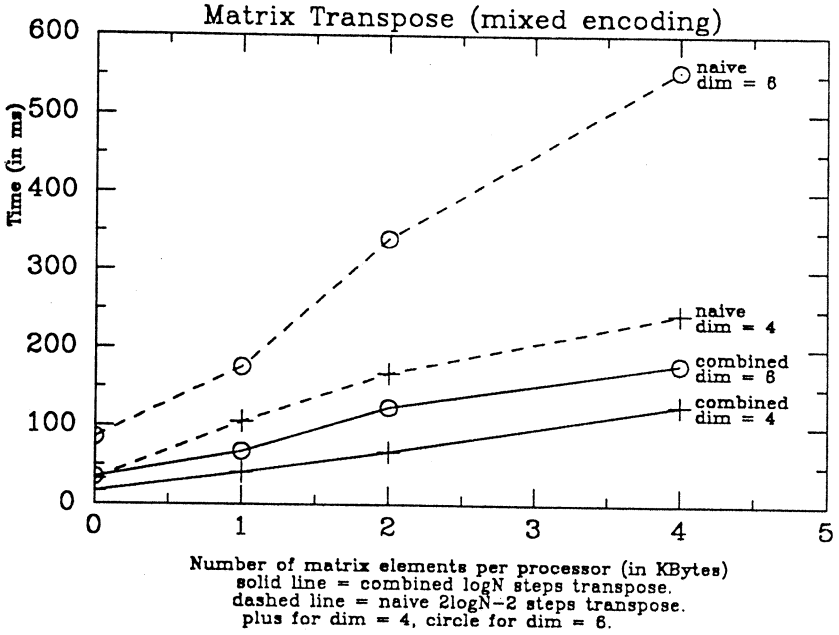

Figure 5: Measured times of transposing a matrix stored by mixed encoding of rows and columns by the naive and combined algorithms on the Intel iPSC.

5. Experiments and Implementation issues

5.1. 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. However, our implementation deviates from the above description in that we do not perform the shuffle operations explicitly, since the copying time on the Intel iPSC is significant. Copying 1024 single precision floating-point numbers (*4k bytes*) takes about 37 milliseconds according to our measurements(Figure 6). Instead, we logically partition the local array into 2^j same-sized blocks during step j. 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 7 presents the measurements for unbuffered and buffered communication for rearrangment of cyclic to consecutive partitioning (one-dimensional local arrays).

The complexity of the unbuffered communication is easily found to be $T = n\frac{PQ}{N}t_c + (N + \lceil\frac{PQ}{2B_mN}\rceil min(n, \log_2\lceil\frac{PQ}{B_mN}\rceil) - \frac{PQ}{B_mN})2\tau$. With buffered communication, messages may intially be larger than the buffer size, in which case they are sent directly. Small messages are buffered and the time for communication is $T = n\frac{PQ}{N}t_c + (\min(n, \log\lceil\frac{PQ}{B_mN}\rceil)\lceil\frac{PQ}{2B_mN}\rceil + \min(N, \frac{PQ}{B_{copy}N}) - \min(N, \frac{PQ}{B_mN}) + \lceil\frac{PQ}{2B_mN}\rceil \max(0, n - \log\lceil\frac{PQ}{B_{copy}N}\rceil))2\tau + \frac{PQ}{N}\max(0, n - \log\lceil\frac{PQ}{B_{copy}N}\rceil)t_{copy}$, 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 7. The

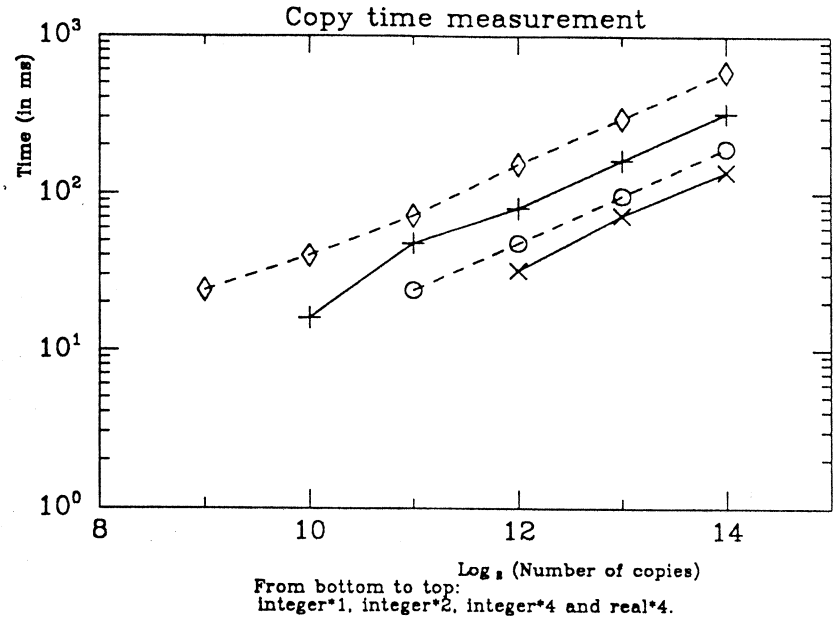

Figure 6: Measured times for copy of various data types on the Intel iPSC.

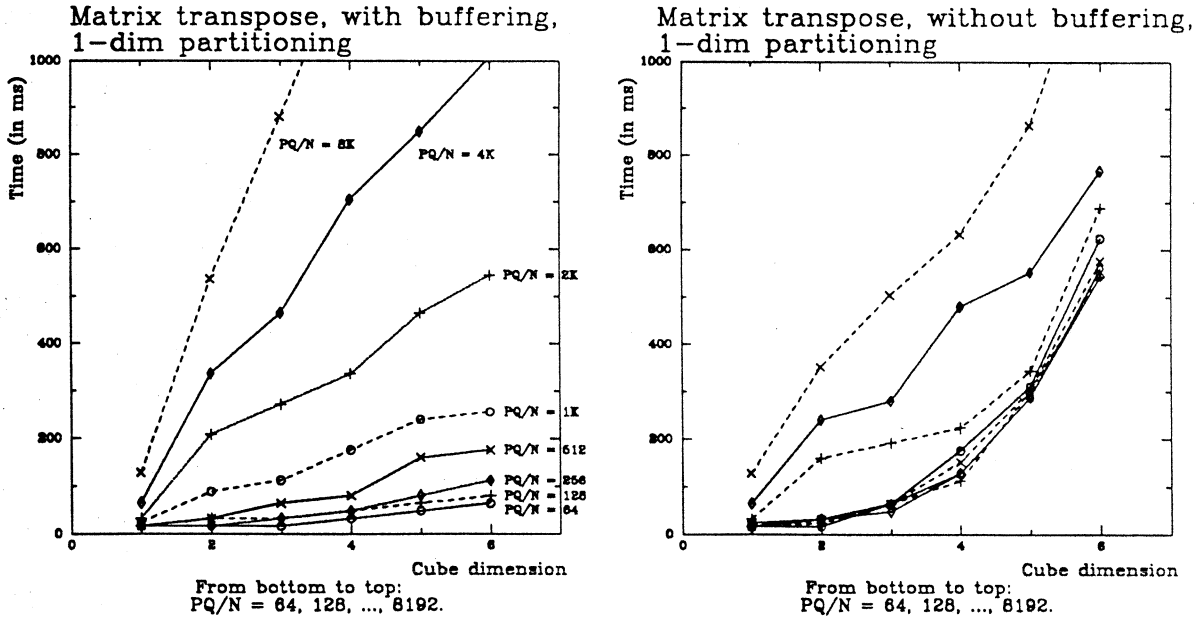

Figure 7: Measured times on the Intel iPSC for the transpose of a matrix, one-dimensional partitioning (or for conversion of consecutive to cyclic one-dimensional partitioning), encoded in binary code.

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 to send blocks of length at least 64 floating-point numbers without buffering. Figure 8 illustrates the sensitivity of the performance to the choice of minimum unbuffered message size. Figure 9 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.

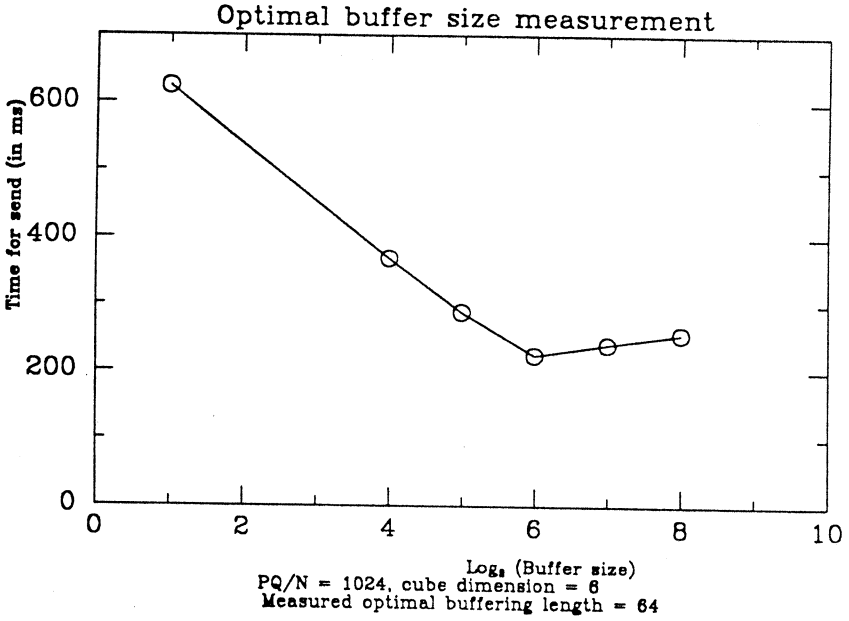

Figure 8: Performance measurements for optimum buffer size on the Intel iPSC.

5.2. Two-dimensional partitioning

5.2.1. The Intel iPSC

We have implemented algorithm *SRT* 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 block transpose of $T = (\frac{PQ}{N} t_c + \lceil \frac{PQ}{B_m N} \rceil \tau) n + 2\frac{PQ}{N} t_{copy}$. The growth rate is proportional to the number of matrix elements. There is an exponential decay as well as a linear increase in the number of cube dimensions. Figure 10 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 11(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

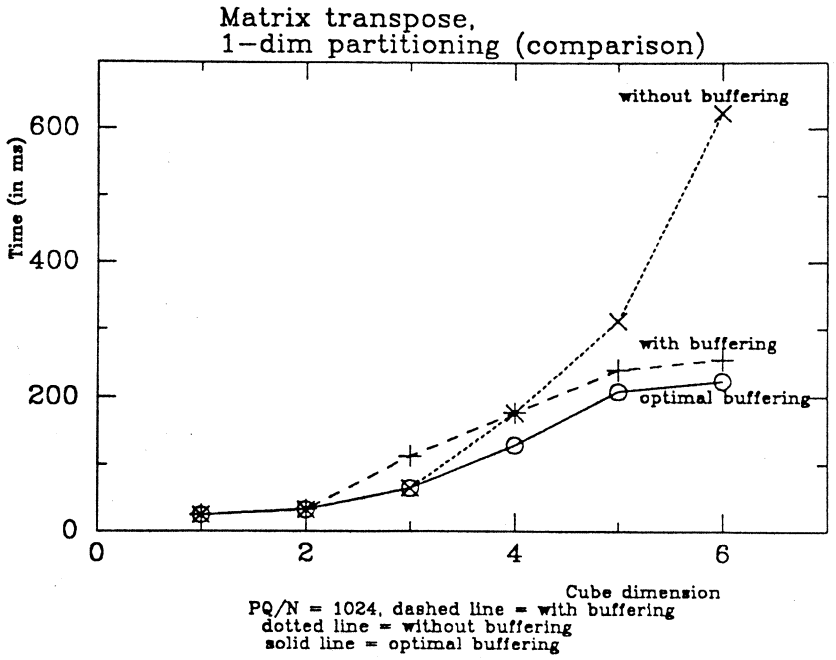

Figure 9: The effect of optimum buffering on performance for matrix transpose on the Intel iPSC.

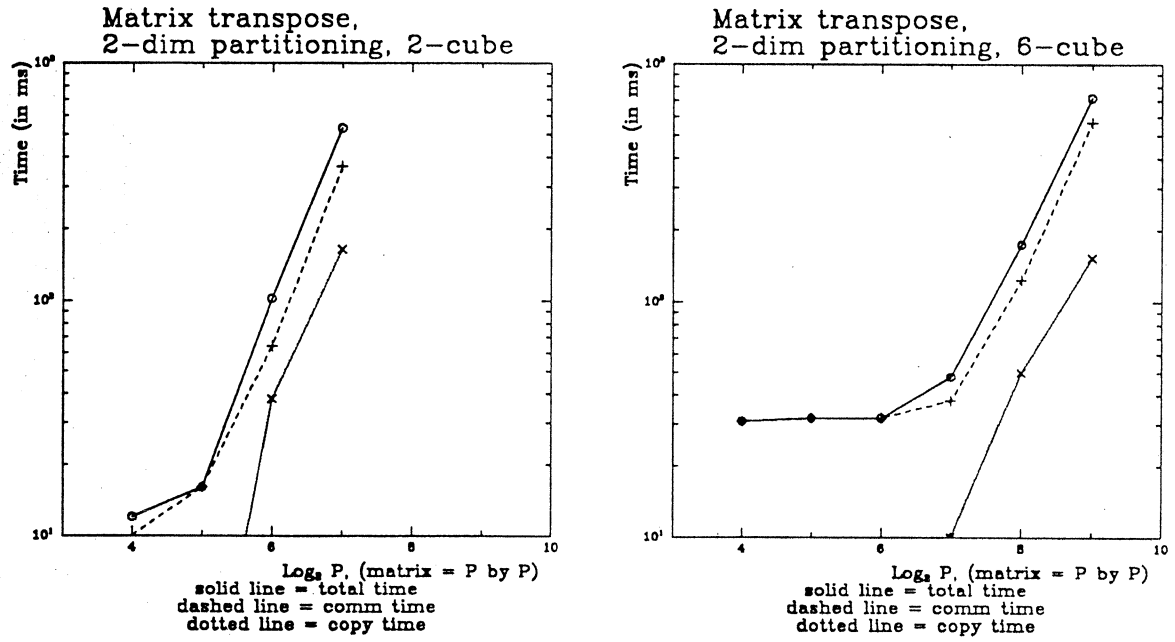

Figure 10: Performance measurements for matrix block transpose on the Intel iPSC.

time increases with the number of cube dimensions, but as the matrix size increases the transpose time decreases with increased cube size.

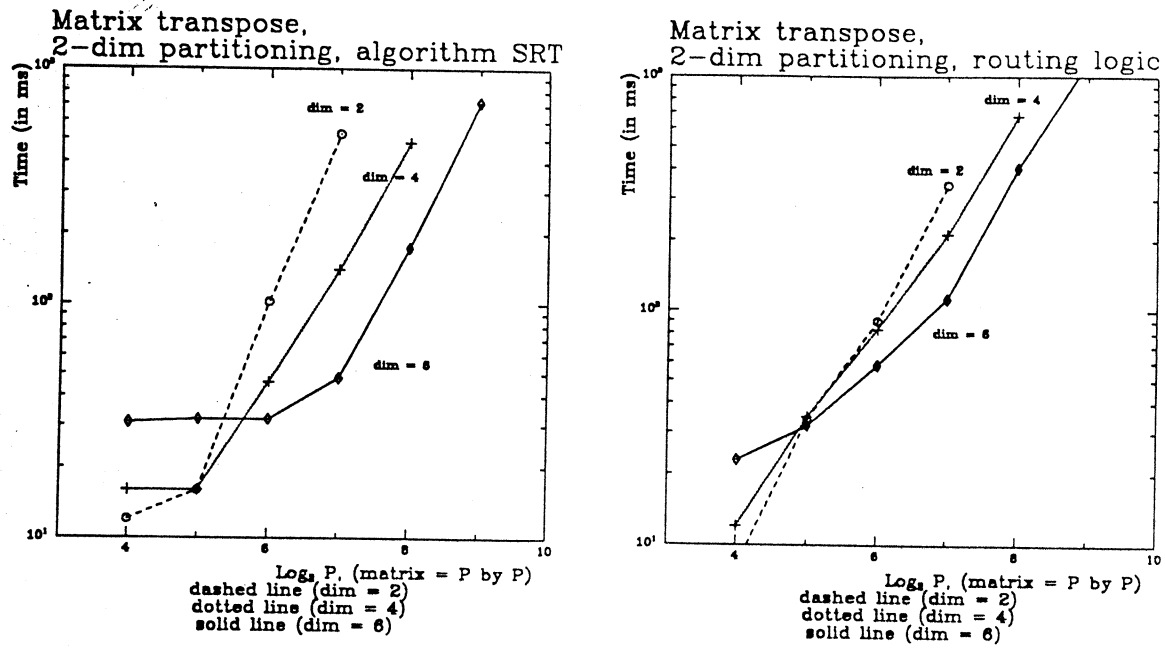

Figure 11: Measured times for block matrix transpose on the Intel iPSC using the *SRT* algorithm without pipelining (a) and using routing logic (b).

On the Intel iPSC it is also possible to carry out the transpose operation by a direct send to the final destination. Figure 11(b) gives the times measured for matrix transpose using the routing logic alone. As the cube size increases the recursive block transpose algorithm yields a significantly better performance than the transpose time offered by the routing logic.

5.2.2. 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 12 shows the transpose time using the routing logic. Each processor holds one matrix element (32-bits). Figure 13 shows the transpose times for various number of matrix elements per processor, and for various number of processors. Figure 14 shows the transpose times for two fixed sized matrices on various sizes of the Connection Machine.

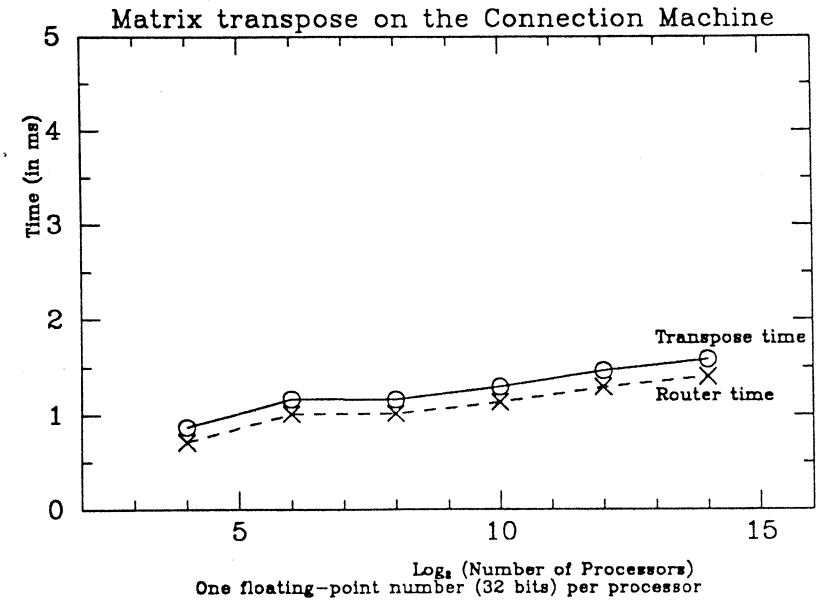

Figure 12: Matrix transpose on the Connection Machine. One element per processor.

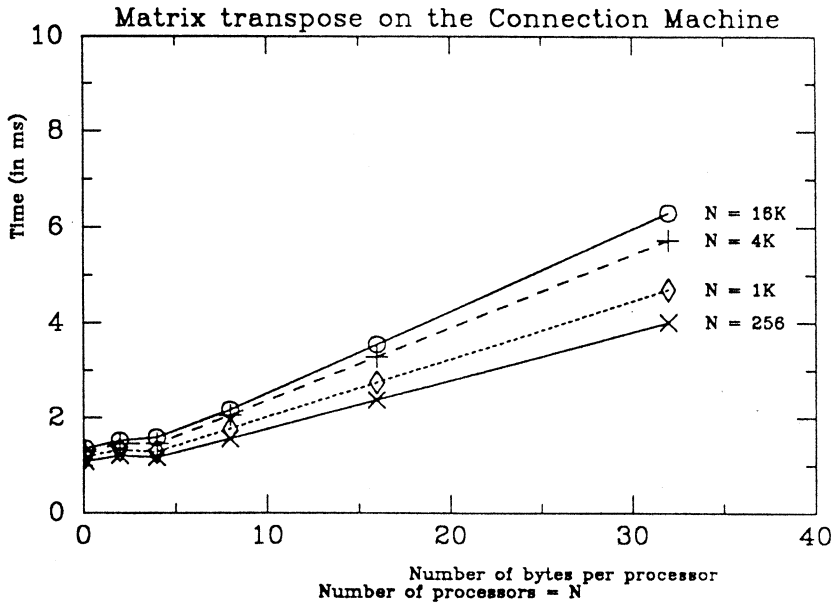

Figure 13: Matrix transpose on the Connection Machine. Multiple elements per processor.

6. 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 block

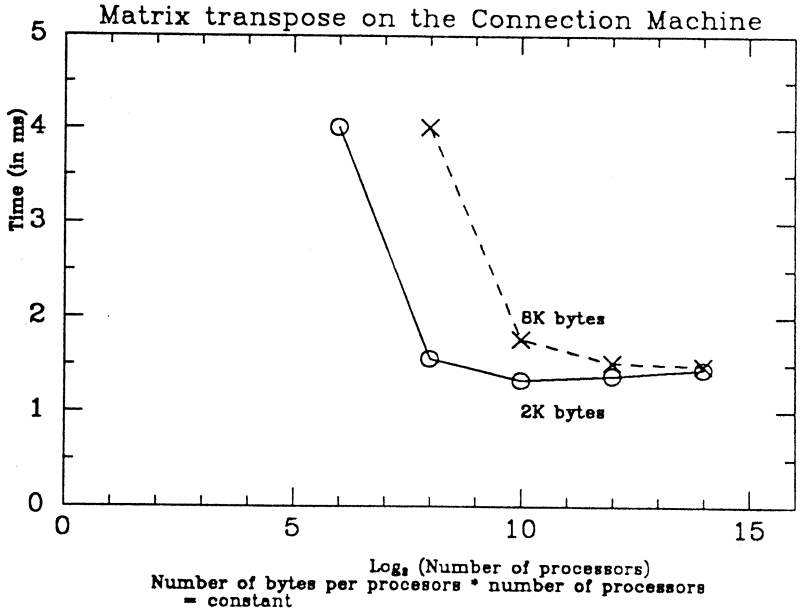

Figure 14: Matrix transpose on the Connection Machine as a function of the machine size.

transpose

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

with that for the strip transpose

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

For $\log\frac{PQ}{B_m N} \geq n$ and $\log\frac{PQ}{B_{copy}N} \geq n$ there is no buffering of communication for the strip transpose. This implies no copy for the strip transpose and a lower communication complexity than for the block transpose. The block transpose always needs an initial and final copy. If the local data structures for the strip transpose are two-dimensional arrays, the same copy operations may be required for both strip and block transpose. But the conclusion remains that for problems which are large relative to the size of the cube the one-dimensional partitioning is most efficient with respect to performance.

For a cube with a size approaching the size of the matrix the copy time for the one-dimensional partitioning grows, and so does the number of start-ups. Both eventually may be higher than that for the two-dimensional transpose. We conclude that for sufficiently large cubes the block transpose is preferable for the Intel iPSC. Figure 15 gives the experimental evidence for this conclusion.

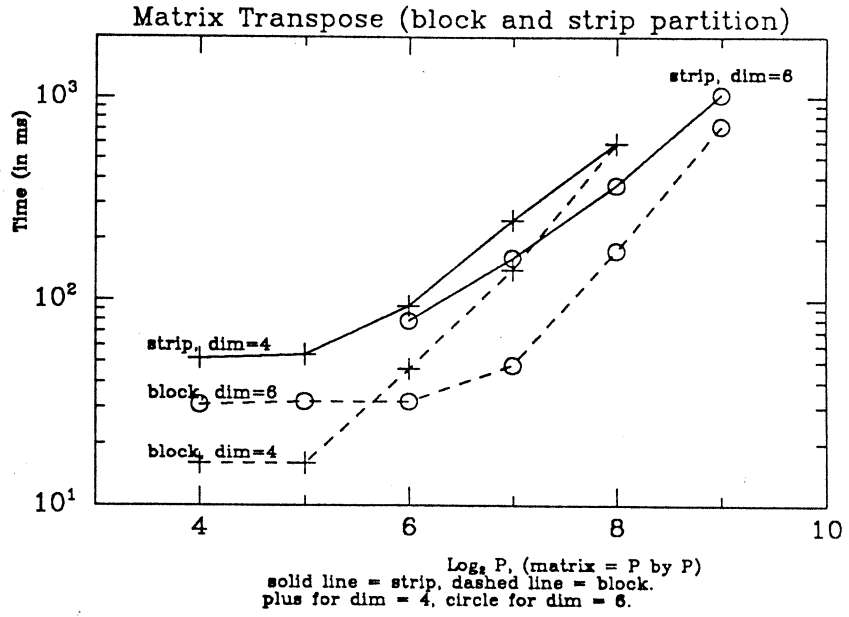

Figure 15: Comparison of the matrix transpose operation of one- and two-dimensional partitioned matrices on the Intel iPSC.

Note that if the copy time can be ignored, then the one-dimensional partitioning always yields a better performance than a two-dimensional partitioning if communication is restricted to one port at a time.

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)

$$T_{min} = \frac{1}{2}(1 - \frac{1}{N})\frac{PQ}{N}t_c + n\tau$$

and

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

and the maximal packet size is

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

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 *Balanced Spanning Tree*(BST) 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

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

References

[1] Eklundh, J.O., *A Fast Computer Method for Matrix Transposing,* IEEE Trans. Computers, C-21/7 (1972), pp. 801–803.

[2] Hillis W.D., *The Connection Machine,* MIT Press, 1985.

[3] Ho C.-T., Johnsson S.L., *Tree Embeddings and Optimal Routing in Hypercubes,* Technical Report YALEU/CSD/RR-, Yale University, Dept. of Computer Science, In preparation 1986.

[4] Johnsson S.L., *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.

[5] ————, *Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures,* Technical Report YALEU/CSD/RR-361, Dept. of Computer Science, Yale University, January 1985.

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

[7] ————, *Solving Tridiagonal Systems on Ensemble Architectures,* SIAM J. Sci. Stat. Comp., (1986). Also available as Report YALEU/CSD/RR-436, November 1985.

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

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

[10] Stone, H.S., *Parallel Processing with the Perfect Shuffle,* IEEE Trans. Computers, C-20 (1971), pp. 153–161.