Optimal Communication Channel Utilization for Matrix Transposition and Related Permutations on Binary Cubes

Optimal Communication Channel Utilization for Matrix Transposition and Related Permutations on Binary Cubes

S. Lennart Johnsson
Ching-Tien Ho

TR-16-92

August 1992

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

This work has in part been supported by the Air Force Office of Scientific Research under contract AFOSR-89-0382, and in part by NSF and DARPA under contract CCR-8908285.

Optimal Communication Channel Utilization for Matrix Transposition and Related Permutations on Binary Cubes

S. Lennart Johnsson
Harvard University and
Thinking Machines Corp.
Cambridge, MA
Johnsson@harvard.edu

Ching-Tien Ho
IBM Almaden Research Center
San Jose, CA 95120
Ho@almaden.ibm.com

Abstract

We present optimal schedules for permutations in which each node sends one or several unique messages to every other node. With concurrent communication on all channels of every node in binary cube networks, the number of element transfers in sequence for K elements per node is $\frac{K}{2}$, irrespective of the number of nodes over which the data set is distributed. For a succession of s permutations within disjoint subcubes of d dimensions each, our schedules yield $\min(\frac{K}{2}+(s-1)d, (s+3)d, \frac{K}{2}+2d)$ exchanges in sequence. The algorithms can be organized to avoid indirect addressing in the internode data exchanges, a property that increases the performance on some architectures.

For message passing communication libraries, we present a blocking procedure that minimizes the number of block transfers while preserving the utilization of the communication channels. For schedules with optimal channel utilization, the number of block transfers for a binary d-cube is d. The maximum block size for K elements per node is $\lceil \frac{K}{2d} \rceil$.

1 Introduction

We give simple, yet optimal, schedules for a class of permutations on multiprocessors configured as binary cubes with concurrent communication on all channels of every processor, *all-port* communication. The class of permutations we consider is all–to–all personalized communication (AAPC), in which each node sends a unique message to every other node. The Connection Machine systems CM–2 and CM–200 [20] are examples of binary cube configured multiprocessor architectures allowing concurrent communication on all channels of every node. Our schedules avoid indirect addressing in the data exchanges by performing a local alignment of the data in each processor prior to, and after, the data interchanges between processors.

Examples of all–to–all personalized communication are bit-reversal, vector-reversal, matrix transposition and shuffle permutations. Conversion between cyclic and consecutive mapping [9] for array allocations, such that a number of memory address bits are exchanged with the same number of processor bits, also constitutes all–to–all personalized communication. Cyclic and consecutive data layout can be specified in Vienna Fortran [21] and in Fortran D [4]. Both layouts are also included in the emerging High Performance Fortran standard. Any one of the above permutations, combined with

code conversion such as conversion between binary code and binary-reflected Gray code [10, 14], also constitutes all–to–all personalized communication.

A succession of AAPCs is necessary in some computations. For instance, a Fast Fourier Transform on 4096 data elements distributed evenly over 512 processors can be performed as a sequence of three local transforms, each on eight data elements. Between successive local transforms, an all–to–all personalized communication within 3-cubes must be performed. Successive AAPCs are performed on successive sets of three processor dimensions. With data in cyclic order [9], the first communication exchanges the local memory address bits (maddr) with the bits AAPC-2 within the processor address field (paddr). The second AAPC exchanges the local memory address bits with the bits AAPC-1, etc.

$$\underbrace{(\underbrace{a_{4d-1}a_{4d-2}\ldots a_{3d}}_{\text{AAPC-2}}\underbrace{a_{3d-1}a_{3d-2}\ldots a_{2d}}_{\text{AAPC-1}}\underbrace{a_{2d-1}a_{2d-2}\ldots a_{d}}_{\text{AAPC-0}}}_{\text{paddr}}|\underbrace{a_{d-1}a_{d-2}\ldots a_0}_{\text{maddr}})$$

Each segment of the processor address field can be viewed as an axis encoded in d bits. The successive AAPCs in the example perform the axis exchanges $(l,k,j|i) \rightarrow (l,k,i|j) \rightarrow (l,j,i|k) \rightarrow (k,j,i|l)$, which constitute a *generalized shuffle* [5, 6, 11].

For the pipelining of a succession of AAPCs, the time elapsed from the first motion of an element until it reaches its destination affects the pipeline delay. We refer to the maximum elapsed time for any element as the *span*. We assume that each node in a d dimensional binary cube, d-cube, concurrently can send and receive one data element on all its ports in one time step, i.e., *all-port* communication. The *all-port* communication implies that all communication links are full duplex. The number of elements per processor is K.

The lower bound for the number of time steps for all-port, all–to–all personalized communication can be shown to be $\frac{K}{2}$ [1, 12]. The optimum span is d. We present five algorithms of which the first three either has an optimum number of element transfers or optimum span. The fourth algorithm has both an optimum number of element transfers and optimum span. The first three algorithms are very simple and provide some of the essential ideas for the fourth algorithm. The fifth algorithm is for multiple, pipelined AAPCs.

In many message-passing libraries for multiprocessors, there is a significant overhead for each message. It is often desirable to send few messages with a large amount of data in each message instead of many messages with a small amount of data in each message. We present a blocking procedure that minimizes the number of block transfers and the block size for a given schedule for a single AAPC. The blocking preserves the number of time steps, i.e., the utilization of the communication links is the same with blocking as without blocking. For Algorithm 4 that has the optimum number of time steps and optimum span, the minimal number of block transfers is d, which is optimal. The block size for this number of block transfers is $\lceil \frac{K}{2d} \rceil$, which is minimal for d block transfers.

The procedure for minimizing the number of messages in a message-passing system can also be used to achieve optimal utilization of communication links when the width of the communication links is a multiple of the element size, or when there are multiple links between pairs of nodes. The Connection Machine systems CM–2 and CM–200 have two communication links between pairs of nodes forming a binary cube with up to 11 dimensions. For a channel width of b elements, our procedure yields $d\lceil \frac{K}{2db} \rceil$ time steps for a single AAPC. Thus, the time is reduced in proportion to the width of each link.

Saad and Schultz [18] have suggested a recursive AAPC algorithm based on 2^d translated binomial trees. The algorithm requires $d\frac{K}{2}$ time steps; it is primarily of interest for *one-port* communication, i.e., communication restricted to one send and one receive per node per time step. *One-port* communication

algorithms have also been presented by Nassimi and Sahni [16, 17] and Flanders [3]. Nassimi and Sahni discuss bit-permute complement (BPC) permutations on mesh or hypercube configured multiprocessor systems with *one-port* communication. Flanders [3] considers similar permutations on two–dimensional meshes with *one-port* communication.

In [12] we presented *one-port* and *all-port* algorithms based on balanced trees and rotated binomial trees. For *all-port* communication, the algorithms attain the optimum number of time steps, when each node sends a multiple of d elements to every other node, i.e., $K = \alpha d 2^d$ for some integer α. For other values of K, the algorithms in [12] are optimal within a small constant factor ($\leq 24\%$ for $d > 4$) [7]. Algorithms 2, 3, and 4 presented here all have the optimum number of time steps for any K. Algorithms 1 and 4 have the optimum span. Independent of the work reported in [12], Stout and Wagar [19] also gave an algorithm with the optimum number of time steps for *all-port* communication. Later, Bertsekas et al. [1] presented yet another algorithm with an optimal number of time steps for any K. A detailed optimal scheduling algorithm for *all-port* communication has also been presented by Edelman [2], who implemented the algorithm on the Connection Machine system CM–2. The issue of minimum span is not addressed in any of the previous algorithms. Indeed, the schedule used by Edelman has a span that is greater than 2^{d-2}. The schedule computation time is proportional to $O(2^d)$. Our Algorithm 4 improves upon previous algorithms by offering both the optimum number of time steps and optimum span, and by easily being amenable to blocking for message-passing communication libraries, while preserving the efficiency in using the communication links. Our Algorithms 2 and 3 have very simple control and are easier to implement than the algorithms in [1, 2].

The outline of this paper is as follows. In Section 2 we define the concepts used and ideas common to the algorithms presented in this paper. We also discuss pipelining of several AAPCs on different sets of cube dimensions, but with a shared set of local memory dimensions. The need for efficient pipelining of a sequence of AAPCs was our motivation for devising Algorithm 4. In Section 3 we present our algorithms for a single AAPC. Section 4 discusses the use of the algorithms for a single AAPC in performing multiple AAPCs. An algorithm performing multiple concurrent AAPCs is presented. Section 5 presents an idea for minimizing the communications overhead in message-passing libraries, and Section 6 presents an idea for the efficient utilization of wide channels. Section 7 gives a summary of our results.

2 All–to–all personalized communication on binary cubes

2.1 Preliminaries

A binary n-cube has $N = 2^n$ nodes. Each node has n neighbors which, with the conventional binary addressing scheme, correspond to the n different single bit complementations of the bits in a node address. For two nodes a and b with addresses $(a_{n-1}a_{n-2}\ldots a_0)$ and $(b_{n-1}b_{n-2}\ldots b_0)$ the Hamming distance is $\sum_{i=0}^{n-1}(a_i \oplus b_i)$, where $\oplus$ denotes modulo two addition. There are n edge-disjoint paths of length n between any pair of nodes at distance n. The local address space in each processing node, or processor for short, is $\mathcal{K} = \{0, 1, \ldots, K-1\}$, and the global address is $(j|i)$, where j is the node address and i the local memory address. The bits required for address encoding are sometimes referred to as dimensions.

Definition 1 Let $\mathcal{S}_s = \{0, 1, \ldots, d-1\}$, $\mathcal{S}_p \subset \{d, d+1, \ldots, d+n-1\}$, where n is the number of cube dimensions. d is the number of cube dimensions involved in the permutation and also the number of local memory dimensions involved in the permutation. Furthermore, let f be a bijection $S_s \rightarrow S_p$ and g be a bijection $S_p \rightarrow S_s$. An AAPC is a permutation defined by $a_i \rightarrow a_{f(i)}$ for all $i \in \mathcal{S}_s$ and

$a_j \rightarrow a_{g(j)}$ for all $j \in \mathcal{S}_p$.

Note that in our definition of the AAPC $|\mathcal{S}_s| = |\mathcal{S}_p|$. $\mathcal{S}_s$ and $\mathcal{S}_p$ represent the set of local memory dimensions and processor dimensions involved in the AAPC, respectively. For simplicity in notation, we assume in the definition that all memory dimensions are involved in the AAPC. An example of AAPCs satisfying definition 1 is the bit-reversal $(a_7a_6a_5a_4a_3|a_2a_1a_0) \rightarrow (a_7a_6a_0a_1a_2|a_3a_4a_5)$, where $n = 5$, $d = 3$, $\mathcal{S}_s = \{0, 1, 2\}$ and $\mathcal{S}_p = \{3, 4, 5\}$. This bit-reversal operation, in fact, consists of four independent bit-reversals in three-dimensional subcubes. The subcubes are identified by the two leading processor address bits. Another example is the axis exchange $(a_6a_5a_4a_3|a_2a_1a_0) \rightarrow (a_2a_1a_0a_3|a_6a_5a_4)$, for which $n = 4$, $d = 3$, $\mathcal{S}_s = \{0, 1, 2\}$ and $\mathcal{S}_p = \{4, 5, 6\}$. This axis exchange can formally be represented as $(k, j|i) \rightarrow (i, j|k)$, where i initially is encoded in the three local memory dimensions $\{0, 1, 2\}$ and k initially is encoded in the processor dimensions $\{4, 5, 6\}$.

If $2^d < K$, then the AAPC is repeated $\lceil \frac{K}{2^d} \rceil$ times. For instance, if $K = 256$ and the AAPC includes three processor dimensions ($d = 3$), then the permutation is repeated $2^{8-3} = 32$ times. For clarity, we assume that $K = 2^d$ in the remainder of this paper. Conversely, if there are processor dimensions not included in the AAPC, then, in fact, a number of AAPCs in disjoint subcubes is specified, as shown in the examples above. The subcube for each AAPC is identified by the processor address bits not included in the specification of the AAPC.

The *relative address* of a local memory address i in processor j with respect to its destination is $j \oplus i$. A *homogeneous* communication *schedule* has schedules for each node that only depend on the *relative addresses*. Thus, in a homogeneous schedule, node i sends data to node $i \oplus j$ in the same dimension and step as node 0 sends data to node j. The message path from node i to node $i \oplus j$ is a *translation* (modulo two addition) with respect to node i of the path from node 0 to node j. We only consider homogeneous schedules. For such schedules, it is sufficient to consider schedules for node 0. All our schedules are based on minimum length path routing of all elements in a node and use *all ports* of every node in every step, except possibly the last step.

Definition 2 Let a data element i leave the source during step t_1 and arrive at the destination during step $t_s > t_1$. Then, $span(i) = t_s - t_1 + 1$. The *span* of an AAPC is the maximum span for any data element, i.e., $span = \max_i span(i)$.

Thus, the *span* for element i is the number of exchanges required for it to move from source to destination, including exchanges during which the element may be waiting en route.

In Algorithm 4 we use the notion of necklaces. A *necklace* [15] is a set of addresses derived from each other through rotations of some fixed bit string. A necklace is *full* if it has d distinct addresses for a string of length d. Otherwise, it is *degenerated*. The *period* of a bit string is the minimum number of rotations (> 0) required to generate a bit string identical to the unrotated string. An address in a degenerate necklace is *cyclic*. Addresses in full necklaces are *noncyclic*. For instance, the address (0000) is cyclic with period one and the address (0101) is cyclic with period two; the addresses (0001) and (0011) are noncyclic. The address with smallest value in the necklace is a *distinguished address*. A q-necklace is a necklace in which each address has q bits equal to 1.

2.2 Algorithm organization

We first consider the organization of algorithms for axes exchanges of the form $(j|i) \rightarrow (i|j)$, then discuss permutations of the form $(j|i) \rightarrow (f(i)|g(j))$.

rel-addr	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	(0|000)	(1|001)	(2|010)	(3|011)	(4|100)	(5|101)	(6|110)	(7|111)
1	(0|001)	(1|000)	(2|011)	(3|010)	(4|101)	(5|100)	(6|111)	(7|110)
2	(0|010)	(1|011)	(2|000)	(3|001)	(4|110)	(5|111)	(6|100)	(7|101)
3	(0|011)	(1|010)	(2|001)	(3|000)	(4|111)	(5|110)	(6|101)	(7|100)
4	(0|100)	(1|101)	(2|110)	(3|111)	(4|000)	(5|001)	(6|010)	(7|011)
5	(0|101)	(1|100)	(2|111)	(3|110)	(4|001)	(5|000)	(6|011)	(7|010)
6	(0|110)	(1|111)	(2|100)	(3|101)	(4|010)	(5|011)	(6|000)	(7|001)
7	(0|111)	(1|110)	(2|101)	(3|100)	(4|011)	(5|010)	(6|001)	(7|000)

Table 1: Data distribution for AAPC in a 3-cube, after the alignment in phase 1.

2.2.1 Axes exchanges: $(j|i) \rightarrow (i|j)$

The permutation $(j|i) \rightarrow (i|j)$, where j and i have the same number of bits, is equivalent to the transposition of a matrix stored with one row per processor to the storage of the matrix with one column per processor. The permutation can also be viewed as changing the allocation of a one-dimensional array from consecutive storage to cyclic storage [9], two storage forms included in Vienna Fortran [21] and Fortran D [4], and adopted in the emerging High Performance Fortran standard.

The routing algorithms we present use only direct addressing in the data exchanges between processors. All processors access the same local memory address during the same step. To accomplish this property, a local alignment precedes the interprocessor exchanges, and a realignment follows the exchanges. On the Connection Machine systems CM–2 and CM–200, avoiding indirect addressing in the exchange phase results in a significant speedup.

Phase 1: Local alignment. Sort the local data by relative address, i.e., perform the operation $(j|i) \rightarrow (j|j \oplus i)$.

Table 1 shows the data distribution in a 3-cube after the alignment. The processor addresses are given in the decimal number system and the local memory addresses in the binary number system.

Phase 2: Interprocessor exchange. The interprocessor exchange phase implements the operation: $(j|i) \rightarrow (j \oplus i|i)$.

Phase 3: Local realignment. The realignment to restore the local memory order is identical to phase 1.

The alignment is performed on the memory dimensions included in the AAPC.

A memory address is subject to an exchange operation in processor dimensions that corresponds to nonzero bits in its relative address. Thus, local memory address zero retains its content throughout phase 2, whereas local memory address $(11\ldots1)$ needs to send its content across all cube dimensions (in any order). Similarly, local memory address $(00\ldots01)$ only exchanges its content with the neighboring node in the least significant cube dimension, while local memory address $(11\ldots10)$ needs to send its content across all but the least significant cube dimension.

In an AAPC, including all local memory dimensions, either local memory address i or $\overline{i}$ needs to send its content across a given cube dimension. Thus, each *complement pair* $(i, \overline{i})$ of local memory addresses must send one element across each cube dimension. The pairing of local memory addresses is shown in Table 2 for $d = 3$.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
(0|000)	(1|001)	(2|010)	(3|011)	(4|100)	(5|101)	(6|110)	(7|111)
(0|111)	(1|110)	(2|101)	(3|100)	(4|011)	(5|010)	(6|001)	(7|000)
(0|001)	(1|000)	(2|011)	(3|010)	(4|101)	(5|100)	(6|111)	(7|110)
(0|110)	(1|111)	(2|100)	(3|101)	(4|010)	(5|011)	(6|000)	(7|001)
(0|010)	(1|011)	(2|000)	(3|001)	(4|110)	(5|111)	(6|100)	(7|101)
(0|101)	(1|100)	(2|111)	(3|110)	(4|001)	(5|000)	(6|011)	(7|010)
(0|011)	(1|010)	(2|001)	(3|000)	(4|111)	(5|110)	(6|101)	(7|100)
(0|100)	(1|101)	(2|110)	(3|111)	(4|000)	(5|001)	(6|010)	(7|011)

Table 2: The initial data distribution viewed as complement pairs of local addresses after alignment in a 3-cube.

2.2.2 Axes exchanges with permutations: $(j|i) \rightarrow (f(i)|g(j))$

The algorithm organization for axes exchanges $(j|i) \rightarrow (i|j)$ is easily generalized to permutations of the form $(j|i) \rightarrow (f(i)|g(j))$, where $f(i)$ is a bijection from S_s to S_s and g a bijection $\mathcal{S}_p \rightarrow \mathcal{S}_s$. The operation $i \rightarrow f(i)$ can be combined with the prealignment operation (phase 1) without a need for additional data motion. Similarly, the permutation $j \rightarrow g(j)$ can be performed as a local memory operation by combining the permutation with the postalignment operation (phase 3). For instance, if the processor addresses are encoded in a binary-reflected Gray code and the local memory addresses use the standard binary code, then an axes exchange preserving the encoding strategy requires no extra data motion. But, the pre– and postalignment phases must be modified to include the code conversion from binary code to binary-reflected Gray code (f), and from binary-reflected Gray code to binary code ($g = f^{-1}$) [10, 14]. The functions f and g adds to the complexity of the address calculation for the local permutations in the pre– and postalignment.

2.3 Multiple AAPC

A sequence of AAPCs can be represented as a sequence of axis exchanges. For instance, in the FFT example given in the introduction, the sequence of AAPCs was represented by the axis exchanges $(l,k,j|i) \rightarrow (l,k,i|j) \rightarrow (l,j,i|k) \rightarrow (k,j,i|l)$.

Lemma 1 *The alignments for AAPCs on different processor bits are independent, and can be performed at once.*

The sequence of AAPCs can therefore be organized into three phases as follows:

Phase 1: Local alignment at once for all axes.

Phase 2: Interprocessor exchanges for the different AAPCs.

Phase 3: Local realignment at once for all axes.

For the three-axes example the operations are

$$(l,k,j|i) \xrightarrow{1} (l,k,j|i \oplus j \oplus k \oplus l),$$

$$(l,k,j|i) \xrightarrow{2} (l,k,i \oplus j \oplus k \oplus l|i),$$

$$(l,k,j|i) \xrightarrow{3} (l, i \oplus j \oplus k \oplus l, j|i),$$
$$(l,k,j|i) \xrightarrow{4} (i \oplus j \oplus k \oplus l, k, j|i),$$
$$(l,k,j|i) \xrightarrow{5} (l,k,j|i \oplus j \oplus k \oplus l).$$

The first step is phase 1, the local prealignment. Phase 2 consists of steps two through four. Each step represents an AAPC within distinct subcubes defined by the different axes (sets of processor dimensions), starting with coordinate axis j. All the communications in the second phase are between elements with the same local memory address. The element initially in location $(l,k,j|i)$ is routed to the final location $(k,j,i|l)$ according to the path:

$$(l,k,j|i) \xrightarrow{1} (l,k,j|i \oplus j \oplus k \oplus l) \xrightarrow{2} (l,k,i|i \oplus j \oplus k \oplus l)$$
$$\xrightarrow{3} (l,j,i|i \oplus j \oplus k \oplus l) \xrightarrow{4} (k,j,i|i \oplus j \oplus k \oplus l) \xrightarrow{5} (k,j,i|l).$$

Table 3 shows the data motion in detail for the permutation $(k,j|i) \rightarrow (j,i|k)$, where each axes is encoded in two bits. The row mod 2 *add* shows the index used for the alignment operations, i.e., $k \oplus j$. Figure 1 shows the motion of one set of data elements. The pairing of data elements is based on the local storage dimensions involved in the AAPCs. The pairing is the same as for a single AAPC.

Phase	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
mod 2 add.	00	01	10	11	01	00	11	10	10	11	00	01	11	10	01	00
Initial	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
	1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
alloc.	2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
after	0	5	10	15	17	20	27	30	34	39	40	45	51	54	57	60
mod 2 add.	1	4	11	14	16	21	26	31	35	38	41	44	50	55	56	61
$\updownarrow$	2	7	8	13	19	22	25	28	32	37	42	47	49	52	59	62
	3	6	9	12	18	23	24	29	33	36	43	46	48	53	58	63
after	0	5	10	15	20	17	30	27	40	45	34	39	60	57	54	51
first	4	1	14	11	16	21	26	31	44	41	38	35	56	61	50	55
AAPC	8	13	2	7	28	25	22	19	32	37	42	47	52	49	62	59
$\leftrightarrow$	12	9	6	3	24	29	18	23	36	33	46	43	48	53	58	63
after	0	17	34	51	20	5	54	39	40	57	10	27	60	45	30	15
second	16	1	50	35	4	21	38	55	56	41	26	11	44	61	14	31
AAPC	32	49	2	19	52	37	22	7	8	25	42	59	28	13	62	47
$\leftrightarrow$	48	33	18	3	36	53	6	23	24	9	58	43	12	29	46	63
after	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
mod 2 add.	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\updownarrow$	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Table 3: The global memory state for an AAPC with explicit alignment.

For a sequence of s AAPCs on d dimensions each, and a total of sd dimensions in the permutation, there are s coordinate axes (each encoded in d bits), and s steps in phase two. The steps are as follows:

$$\begin{aligned}
(a_s, a_{s-1}, \ldots, a_1|a_0) &\xrightarrow{1} (a_s, a_{s-1}, \ldots, a_1|a_s \oplus a_{s-1} \oplus \cdots \oplus a_1 \oplus a_0) \\
(a_s, a_{s-1}, \ldots, a_1|a_0) &\xrightarrow{2} (a_s, a_{s-1}, \ldots, a_2, a_s \oplus a_{s-1} \oplus \cdots \oplus a_1 \oplus a_0|a_0) \\
&\ldots
\end{aligned}$$

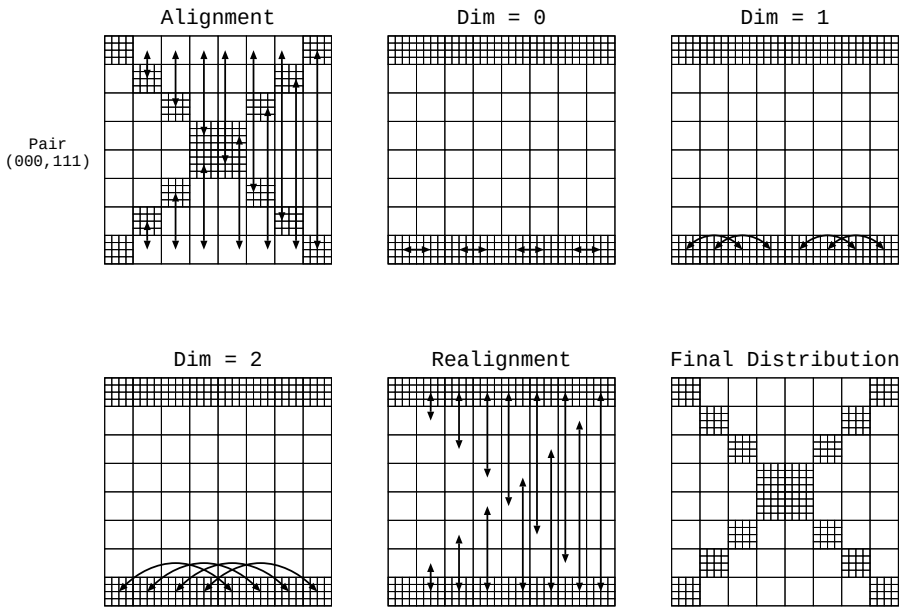

Figure 1: Tracing a set of memory locations for an AAPC with explicit alignment.

$$(a_s, a_{s-1}, \ldots, a_1 | a_0) \overset{s+1}{\rightarrow} (a_s \oplus a_{s-1} \oplus \cdots \oplus a_1 \oplus a_0, a_{s-1}, a_{s-2}, \ldots, a_1 | a_0)$$
$$(a_s, a_{s-1}, \ldots, a_1 | a_0) \overset{s+2}{\rightarrow} (a_s, a_{s-1}, \ldots, a_1 | a_s \oplus a_{s-1} \oplus \cdots \oplus a_1 \oplus a_0)$$

The communication for successive AAPCs can be pipelined. Once a pair of local memory addresses has completed the communication for one AAPC, they are ready to proceed to the next AAPC. Minimizing the pipeline delay is equivalent to minimizing the span. In Section 4 we also show how the pipeline delay can be reduced by performing multiple AAPCs concurrently.

3 Algorithms for a single AAPC

The first three algorithms in this section either have optimum data transfer time, or optimum span, but not both. Algorithm 4, the major contribution of this section, has both optimum data transfer time and optimum span, but is more complex.

3.1 Algorithm 1

A straightforward algorithm for AAPC is to schedule d complement pairs of addresses concurrently, by scheduling complement pair u in dimensions $u, (u+1) \bmod d, (u+2) \bmod d, \ldots, (u+d-1) \bmod d$. After d exchanges, the d pairs of addresses have performed all necessary communications. In each step exactly one of the two addresses in a pair must exchange its content with another processor. The address sending its data to another processor has the relative address bit equal to one for the dimension being exchanged. Table 4 shows the exchanges for the first three complement pairs of addresses for $d = 3$. For the first row of complement pairs of addresses in Table 4, the exchange sequence with respect to processor dimensions is 0, 1, and 2. The sequence for the second row of complement pairs of addresses is 1, 2, and 0. The sequence for the third row of complement pairs of addresses is 2, 0, and 1. Every channel is used in every step; each item follows a minimum-length path.

Time	Dim.	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	0	(0|000)	(1|001)	(2|010)	(3|011)	(4|100)	(5|101)	(6|110)	(7|111)
		(1|110)	(0|111)	(3|100)	(2|101)	(5|010)	(4|011)	(7|000)	(6|001)
	1	(0|001)	(1|000)	(2|011)	(3|010)	(4|101)	(5|100)	(6|111)	(7|110)
		(2|100)	(3|101)	(0|110)	(1|111)	(6|000)	(7|001)	(4|010)	(5|011)
	2	(0|010)	(1|011)	(2|000)	(3|001)	(4|110)	(5|111)	(6|100)	(7|101)
		(4|001)	(5|000)	(6|011)	(7|010)	(0|101)	(1|100)	(2|111)	(3|110)
2	1	(0|000)	(1|001)	(2|010)	(3|011)	(4|100)	(5|101)	(6|110)	(7|111)
		(3|100)	(2|101)	(1|110)	(0|111)	(7|000)	(6|001)	(5|010)	(4|011)
	2	(0|001)	(1|000)	(2|011)	(3|010)	(4|101)	(5|100)	(6|111)	(7|110)
		(6|000)	(7|001)	(4|010)	(5|011)	(2|100)	(3|101)	(0|110)	(1|111)
	0	(0|010)	(1|011)	(2|000)	(3|001)	(4|110)	(5|111)	(6|100)	(7|101)
		(5|000)	(4|001)	(7|010)	(6|011)	(1|100)	(0|101)	(3|110)	(2|111)
3	2	(0|000)	(1|001)	(2|010)	(3|011)	(4|100)	(5|101)	(6|110)	(7|111)
		(7|000)	(6|001)	(5|010)	(4|011)	(3|100)	(2|101)	(1|110)	(0|111)
	0	(1|000)	(0|001)	(3|010)	(2|011)	(5|100)	(4|101)	(7|110)	(6|111)
		(6|000)	(7|001)	(4|010)	(5|011)	(2|100)	(3|101)	(0|110)	(1|111)
	1	(2|000)	(3|001)	(0|010)	(1|011)	(6|100)	(7|101)	(4|110)	(5|111)
		(5|000)	(4|001)	(7|010)	(6|011)	(1|100)	(0|101)	(3|110)	(2|111)

Table 4: The locations of the contents of the first three complement pairs of local addresses after each exchange step in a 3-cube. The position of the entry in the column "Dim" (top or bottom) indicates the address of the data element subject to exchange.

The pseudocode below serves to illustrate the simplicity in implementing the algorithm for phase two of an AAPC performed on the d least significant processor dimensions. The innermost loop performs the concurrent exchanges in all processor dimensions. Index u enumerates different complement pairs of local memory addresses in a concurrent exchange. Index r enumerates the exchanges required for each complement pair of memory addresses. Loop index q enumerates different sets of d complement pairs of addresses, while the forall statement specifies all binary cube processors. The memory accesses over time and the loop ordering are illustrated in Figure 2 for $d = 5$.

By unrolling the loop on q in Algorithm 1, the exchanges corresponding to a few diagonal blocks in Figure 2 can be made in the same exchange step, thereby reducing the number of exchange steps and potentially the overhead in a message-passing communication library. The number of element transfers in sequence is unaffected by such a schedule change. Blocking of element exchanges is discussed further in Section 5.

forall $p \in \{0, 1, \ldots, 2^n - 1\}$ **do**
  for $q := 0$ **to** $\lfloor \frac{2^d}{2d} \rfloor - 1$ **step** d **do**
    for $r := 0$ **to** $d - 1$ **do**
      forall $u \in \{0, 1, \ldots, d - 1\}$ **do**
        if $(q + u)_{(u+r) \bmod d} = 1$ **then**
          $(p_{n+d-1}p_{n+d-2} \cdots p_{(u+r) \bmod d+1}p_{(u+r) \bmod d}p_{(u+r) \bmod d-1} \cdots p_d | q + u) \rightarrow$
          $(p_{n+d-1}p_{n+d-2} \cdots p_{(u+r) \bmod d+1}\overline{p_{(u+r) \bmod d}}p_{(u+r) \bmod d-1} \cdots p_d | q + u)$
        else
          $(p_{n+d-1}p_{n+d-2} \cdots p_{(u+r) \bmod d+1}p_{(u+r) \bmod d}p_{(u+r) \bmod d-1} \cdots p_d | \overline{q + u}) \rightarrow$
          $(p_{n+d-1}p_{n+d-2} \cdots p_{(u+r) \bmod d+1}\overline{p_{(u+r) \bmod d}}p_{(u+r) \bmod d-1} \cdots p_d | \overline{q + u})$
        endif
      endforall
    endfor
  endfor
  ...
  Similar exchanges for the remaining $2^d \bmod 2d$ local addresses.
  ...
endforall

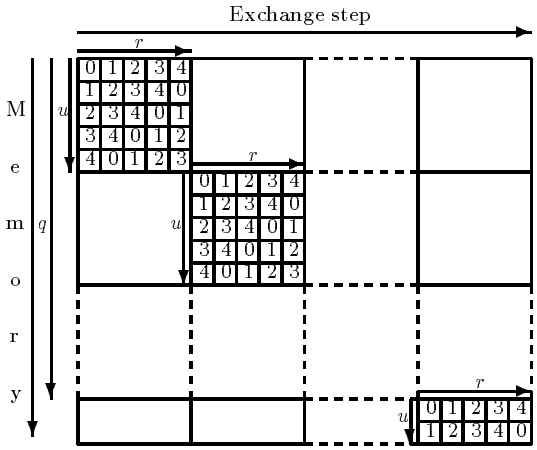

Figure 2: Loop ordering for Algorithm 1.

The correctness of the above algorithm follows from the following consideration:

$$\begin{cases} \text{Phase 1:} & (j|i) \xrightarrow{1} (j|j \oplus i), \\ \text{Phase 2:} & (j|i) \xrightarrow{2} (j \oplus i|i), \\ \text{Phase 3:} & (j|i) \xrightarrow{3} (j|j \oplus i), \end{cases}$$

$$\text{or } (j|i) \xrightarrow{1} (j|j \oplus i) \xrightarrow{2} ((j \oplus i) \oplus j|j \oplus i) = (i|j \oplus i) \xrightarrow{3} (i|(j \oplus i) \oplus i) = (i|j).$$

Phase 2 of Algorithm 1 proceeds in $\lceil \frac{K}{2d} \rceil$ subphases with d exchanges per subphase. If d does not divide $\frac{K}{2}$, then the last subphase (d communication steps) does not fully use all communication channels.

3.2 Algorithm 2

Table 5 defines Algorithm 2. Complement pair u of local memory addresses is exchanged in processor dimension r during time step $(u + r) \bmod \frac{2^d}{2}$, $0 \leq r < d$ and $0 \leq u < \frac{K}{2}$. The span is $\frac{K}{2}$. The AAPC schedule in [2] has a span of order $O(2^d)$ and optimal channel utilization. The large span is a disadvantage when several AAPCs on different processor address bits must be performed.

3.3 Algorithm 3

Optimal channel utilization with a span of $d + \frac{K}{2} \bmod d$ is easily obtained by applying Algorithm 2 to $d + \frac{K}{2} \bmod d$ pairs of memory addresses and Algorithm 1 to all other pairs of memory addresses. We refer to this combined algorithm as Algorithm 3.

3.4 The communication complexity of Algorithms 1 – 3

The characteristics of the above algorithms are summarized in Table 6. The algorithms either has optimum span or an optimum number of time steps, but not both.

Memory loc/pair	Exchange step							
	1	2	3	4	5	6	7	8
(0000, 1111)	0	1	2	3	-	-	-	-
(0001, 1110)	-	0	1	2	3	-	-	-
(0010, 1101)	-	-	0	1	2	3	-	-
(0011, 1100)	-	-	-	0	1	2	3	-
(0100, 1011)	-	-	-	-	0	1	2	3
(0101, 1010)	3	-	-	-	-	0	1	2
(0110, 1001)	2	3	-	-	-	-	0	1
(0111, 1000)	1	2	3	-	-	-	-	0

Table 5: A schedule for optimal channel utilization (Algorithm 2) for an AAPC on five processor dimensions.

Algorithm	Element transfers	Span		Local memory addresses in transit	
		Min	Max	Min	Max
1	$d\lceil\frac{K}{2d}\rceil$	d	d	$2(\frac{2^d}{2} \bmod d)$	$2d$
2	$\frac{K}{2}$	d	$\frac{K}{2}$	$2d$	$2d$
3	$\frac{K}{2}$	d	$d + (\frac{2^d}{2} \bmod d)$	$2d$	$2(d + (\frac{2^d}{2} \bmod d))$

Table 6: The number of element transfers, the span, and the number of local memory addresses scheduled together in Algorithms $1-3$ on a d-cube.

Performing s AAPCs in sequence by pipelining successive applications of Algorithm 1 requires $d\lceil\frac{K}{2d}\rceil + (s-1)d$ time steps. For Algorithm 3, pipelining yields $\frac{K}{2} + (s-1)(d + \frac{K}{2} \bmod d)$ time steps. The schedule devised below (Algorithm 4) has optimal channel utilization and a span of d. Pipelining s AAPC based on this algorithm yields $\frac{K}{2} + (s-1)d$ time steps.

3.5 Algorithm 4

Algorithms 1, 2 and 3 schedule complement pairs of local addresses together. In Algorithm 4 we supplement this scheduling strategy with one based on whether or not an address is *noncyclic*. We first consider a possible scheduling of noncyclic addresses, then a possible scheduling of cyclic addresses. Considering noncyclic and cyclic addresses separately would yield an algorithm with optimum span, but in general nonoptimum data transfer time. By scheduling some cyclic addresses together with some noncyclic addresses, an algorithm with both optimum span and optimum data transfer time can be devised.

3.5.1 Scheduling of noncyclic addresses

The exchanges for the three local memory locations (001), (010), and (100) can be done concurrently. Similarly, locations (011), (110), and (101) can be scheduled concurrently with respect to the least significant bit of (011), the second least significant bit of (110), and the most significant bit of (101). The communication for the remaining bit that is one in each of these addresses can also be scheduled concurrently. In general, for a full q-necklace with bits $i_0, i_1, \ldots, i_{q-1}$ equal to one, the contents of the distinguished address is exchanged in dimensions $i_0, i_1, \ldots, i_{q-1}$. Within the q-necklace, address r

obtained through an r steps left cyclic rotation of the distinguished address, is subject to exchanges in dimensions $(i_0 + r) \bmod d, (i_1 + r) \bmod d, \ldots, (i_{q-1} + r) \bmod d$. For the 1-necklace with distinguished address (001), the described schedule yields a single exchange in dimension $i_0 = 0$ for the distinguished address. For address r in the 1-necklace, the exchange is in dimension r. For the 3-necklace $\{0111, 1110, 1101, 1011\}$, the schedule yields the exchanges

Address	Exchange dimensions		
0111:	0	1	2
1110:	1	2	3
1101:	2	3	0
1011:	3	0	1

Lemma 2 *The set of addresses forming a full q-necklace can be scheduled to complete the required permutation in q exchange steps, which is optimal. The span is $q \leq d$, and all communication channels are used in each exchange step.*

The presented schedule for noncyclic addresses yields optimum span and full utilization of all communication channels. Cyclic addresses cannot be scheduled optimally using the same idea. In our Algorithm 4, we schedule one suitably chosen full necklace of addresses together with a set of complement pairs of cyclic addresses. All other noncyclic addresses are scheduled as described above.

Before considering the scheduling of cyclic addresses, we compare the scheduling of noncyclic addresses based on the idea of necklaces with scheduling based on complement pairs of addresses.

Lemma 3 *Addresses of a q-necklace obtained through bit-complementation of a full $(d-q)$-necklace is also a full necklace, and distinct, if $q \neq d/2$.*

Corollary 1 *The addresses in any full q-necklace, $q \neq d/2$, and its complement $(d-q)$-necklace, can be scheduled together to complete the required data motion in d exchange steps.*

Thus, for d odd, schedules of noncyclic addresses either based on necklaces, or complement pairs of addresses, both yield optimum utilization of all d communication channels. The maximum span for scheduling based on necklaces is $d-1$, while the span for schedules based on complement pairs is d. For $q = d/2$ not every q-necklace has a matching complement necklace. For instance, the address (0011) and its complement (1100) belong to the same necklace. Thus, for d even, scheduling based on complement pairs of addresses may not yield full utilization of all communication channels for all noncyclic addresses.

3.5.2 Scheduling of cyclic addresses

Cyclic addresses are scheduled as complement pairs of addresses. The bit-complement of a cyclic address is also cyclic. Scheduling cyclic addresses as complement pairs of addresses yields full utilization of all d communication channels for all such pairs only when the number of cyclic addresses is a multiple of $2d$.

Let the number of pairs of cyclic addresses be p and let $c = p \bmod d$. Then, all but c pairs of cyclic addresses are scheduled as blocks of d complement pairs in exchange sequences with span d, utilizing

all d communication channels of every processor in every exchange step. The remaining c complement pairs of cyclic addresses are scheduled together with the addresses of one full $(d-c)$-necklace. For example, for $d = 3$ $c = 1$ and the complement pair of cyclic addresses (000) and (111) are scheduled together with the 2-necklace (011), (110) and (101). The remaining addresses all belong to the 1-necklace (001), (010), and (100), which is scheduled as a full necklace with all exchanges completed during a single cycle. The schedule for the cyclic addresses and the 2-necklace are shown below.

Address	Exchange dimensions		
011:	0	1	-
110:	1	-	2
101:	-	2	0
(000,111):	2	0	1

3.5.3 The complete algorithm

We are now ready to define Algorithm 4 as follows:

- schedule blocks of d complement pairs of cyclic addresses together,
- schedule the remaining c complement pairs of cyclic addresses with addresses of a full $(d-c)$-necklace,
- schedule all remaining noncyclic addresses by scheduling the d members of each remaining full necklace together.

It follows from the previous discussion that this algorithm fully utilizes all communication channels of every processor during every exchange step and has a span of d.

We construct the schedule for the c pairs of cyclic addresses that are combined with the addresses in a full $(d-c)$-necklace from a $d \times d$ *generating table*. The columns of the table correspond to exchange steps, and the table entries correspond to the dimensions scheduled for that time step. The task is to associate d of the $d+2c$ local memory addresses that must be scheduled in every exchange step to the d table entries in a column. The first row of the table consists of the numbers $0, 1, \ldots, d-1$. Each other row is a one step left cyclic rotation of the previous row. Thus, every dimension is used for every exchange step. We arbitrarily associate the last c rows of the generating table with the c complement pairs of cyclic addresses. Each complement pair is clearly subject to an exchange in every dimension during d exchange steps, as required. The remaining $d-c$ rows of the table are used to create a schedule for the d addresses of the full $(d-c)$-necklace.

We choose the full $(d-c)$-necklace to be the necklace with distinguished address $(0^c 1^{d-c})$, where 1^{d-c} denotes $d-c$ consecutive 1-bits. There exist several schedules with span d and an optimal number of element transfers. Any valid schedule must associate an address of the necklace with at most one entry in a column and precisely $d-c$ columns. Furthermore, all table entries associated with an address must be unique, and the set of table entries for different addresses must be rotations of each other. Each table entry can only be assigned to one address. The schedules can be illustrated by drawing lines in the generating table. Each line intersects d column entries and represents all addresses in the full $(d-c)$-necklace. Each entry on the line represents an address, and the table entry gives the dimension in which that address is exchanged during the time step given by the table column. In total, $d-c$ lines must be drawn since each address of the $(d-c)$-necklace must be scheduled in $d-c$ dimensions.

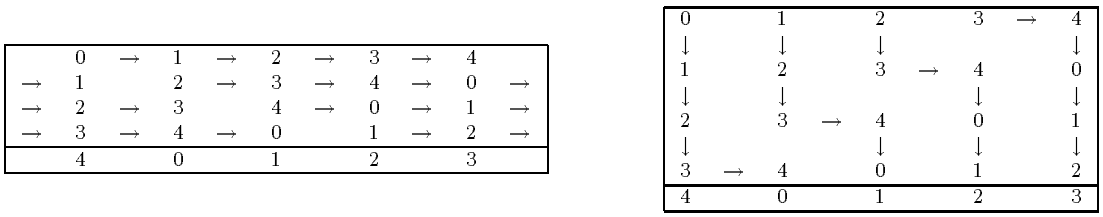

Figure 3: Scheduling lines in the generating table for $d = 5$ ($d - c = 4$).

Address	Time step				
	0	1	2	3	4
10111	0	2	4	1	-
01111	-	1	3	0	2
11110	3	-	2	4	1
11101	2	4	-	3	0
11011	1	3	0	-	4
(00000,11111)	4	0	1	2	3

Address	Time step				
	0	1	2	3	4
01111	0	1	2	3	-
11110	1	2	3	-	4
11101	2	3	-	4	0
11011	3	-	4	0	1
10111	-	4	0	1	2
(00000,11111)	4	0	1	2	3

Table 7: Exchange dimension for type-2 scheduling for $d = 5$ and $q = 4$.

Figure 3 gives the generating table for $d = 5$ with two different schedules with arrows representing the lines. In the left half of the figure, the lines start on the diagonal and progress to the right within a row, cyclicly. The dimensions associated with entry j, $0 \leq j < d$ of line i, $0 \leq i < d - c$, is $(2i + j) \bmod d$, by construction of the generating table and the lines. Thus, for any j, all $d - c$ entries for $0 \leq i < d - c$ are unique for d odd. Member j of the necklace has the address bits set which correspond to the table entries for j on the different lines. Thus, for $d = 5$ $d - c = 4$ and for $j = 0$, $j = 1$, etc., the addresses are {(10111), (01111), (11110), (11101), (11011)}. For d even, the same scheme can be used for $c \geq d/2$. For $c < d/2$ the first $d/2$ lines can be drawn as for d odd, while a skew of two columns is used for line $i = d/2$ in order to preserve uniqueness of all entries for j.

The schedule corresponding to the right half of Figure 3 is obtained by drawing the lines vertically in a "greedy" manner. Thus, the first line is drawn from the top of the first column down to the $(d-c)$th row, then c columns to the right. The second line starts in the second column, goes vertically through $d - c - 1$ rows, then turns right through c columns, followed by a vertical downturn to include one more row. Subsequent lines are drawn in a similar way by proceeding vertically one row less than the previous line, then proceeding horizontally through c columns, then proceeding vertically to the last row reserved for the full necklace.

The schedules corresponding to Figure 3 are shown in Table 7. The first row corresponds to $j = 0$, the second row to $j = 1$, etc.

In summary, in Algorithm 4, local memory addresses are partitioned into cyclic and noncyclic addresses. The number of cyclic addresses is two if d is prime; otherwise it is of order $O(\sqrt{2^d})$ [8]. Cyclic addresses are paired through bit-complementation and divided into blocks of d pairs. The remaining c pairs are scheduled together with $d - c$ noncyclic addresses forming a $(d-c)$-necklace. All remaining noncyclic local memory addresses can be scheduled as blocks of d complement pairs of addresses.

Theorem 1 *An all–to–all personalized communication on d processor dimensions can be performed in* $\frac{K}{2}$ *time steps for all–port communication. The maximum span is d.*

4 Algorithms for multiple AAPCs

For multiple AAPCs with the same set of local memory dimensions, pipelining can be applied to increase the utilization of the communication channels in *all-port* communication. If each AAPC fully utilizes the communication channels, then the only source of inefficiency is the pipeline delay. If there are additional storage dimensions, then the pipeline delay may be avoided by performing multiple, concurrent AAPCs. Below we first comment on the pipelined approach, then outline an algorithm for multiple concurrent AAPCs (Algorithm 5).

4.1 Pipelining of AAPCs

Pipelining the communications for a succession of AAPCs is conceptually straightforward when all the AAPCs involve the same number of local memory and processor dimensions. The communication complexity for pipelined AAPCs follows from Theorem 1.

Corollary 2 *The number of time steps for s all–to–all personalized communications in succession on different sets of d processor dimensions, and the same set of local memory dimensions, is $\frac{K}{2} + (s-1)d$ for a local data set of size K, and all–port communication.*

If the succession of AAPCs shall be performed on nonoverlapping subcubes of different order, then the situation is more complex. If the subcubes are of nonincreasing order, and the local memory bits for AAPCs on smaller subcubes form a subset of the d_1 bits for the first AAPC, then an extra delay is introduced whenever the number of dimensions in an AAPC does not divide the number of dimensions in the previous AAPC. Assume that the number of dimensions for the ith AAPC is d_i. Then, the extra delay introduced by the ith AAPC in the case of Algorithm 1 is $d_i - \gcd(d_{i-1}, d_i)$. No realignment or change of pairing is required for Algorithm 1. For Algorithm 4, a change in the number of local memory dimensions involved in the AAPC affects the classification of addresses as cyclic and noncyclic, and hence their schedules. The regrouping for the ith AAPC may involve addresses from two blocks each of which requires d_{i-1} exchanges steps. Slightly more efficient, and simpler algorithms for this case, are contained in [13].

4.2 Multiple concurrent AAPCs

Consider the exchange sequence $(l,k,j|i,v) \rightarrow (l,v,j|i,k) \rightarrow (k,v,j|i,l) \rightarrow (k,v,j|l,i) \rightarrow (k,v,i|l,j) \rightarrow (k,j,i|l,v)$. The resulting permutation is the same as for the exchange sequence $(l,k,j|i) \rightarrow (l,k,i|j) \rightarrow (l,j,i|k) \rightarrow (k,j,i|l)$. Thus, if there are local memory dimensions in addition to the ones included in the AAPC's, then the first AAPC can be performed on a processor axis other than the first, at the expense of one local memory reordering and of one repeated AAPC. The idea is illustrated in Figure 4 for six processor axes and one local memory axis.

In Figure 4, each column represents d exchanges (as required for any pair of memory addresses). The number of memory addresses associated with a row in the figure depends upon the algorithm used for each AAPC. For Algorithm 1, $2d$ addresses are associated with each table row (compare Figure 2). For Algorithm 4, a table row corresponds to $2d$ memory addresses, except for the noncyclic addresses scheduled together with some of the cyclic addresses. The number of local memory addresses associated with this row is $d + 2c$. The data sets $d_0^{\{0,1\}}$ and $d_1^{\{0,1\}}$ have their first axes exchanges in dimensions other than the first. With the axes labeled from 0, the data sets $d_0^{\{0,1\}}$ perform their first exchange with processor axis $s-2$, while the data sets $d_1^{\{0,1\}}$ perform their first exchanges on processor

	Time step											
Memory	d_0^0	$v,5$	$v,6$	$\bullet$			$i,1$	$i,2$	$i,3$	$i,4$	$i,5$	
	d_0^1		$v,5$	$v,6$				$i,1$	$i,2$	$i,3$	$i,4$	$i,5$
	d_1^0	$v,3$	$v,4$	$v,5$	$v,6$	$\bullet$			$i,1$	$i,2$	$i,3$	
	d_1^1		$v,3$	$v,4$	$v,5$	$v,6$				$i,1$	$i,2$	$i,3$
	d_2	$i,1$	$i,2$	$i,3$	$i,4$	$i,5$	$i,6$					
	d_3		$i,1$	$i,2$	$i,3$	$i,4$	$i,5$	$i,6$				
	d_4			$i,1$	$i,2$	$i,3$	$i,4$	$i,5$	$i,6$			
	d_5				$i,1$	$i,2$	$i,3$	$i,4$	$i,5$	$i,6$		
	d_6					$i,1$	$i,2$	$i,3$	$i,4$	$i,5$	$i,6$	

Figure 4: The exchange sequences for six AAPCs in sequence using arbitrary starting dimensions and pipelining.

axis $s-4$. The $\bullet$ represents the exchange between the two memory axes initially represented by i and v, and corresponds to the exchange $(k,v,j|i,\ell) \rightarrow (k,v,j|\ell,i)$ in the example above.

Algorithm 5 for multiple AAPCs divide the local data sets into groups scheduled together according to a suitable algorithm for a single AAPC. Then, $s-2$ such data sets for s even, and $s-1$ data sets for s odd, have their first exchange on an axis other than the first. Pairs of data sets have their first exchange on the same axis, and perform a local exchange prior to the exchange on the first processor axis. Data sets having their first exchange on the first processor axis have their exchanges pipelined.

For multiple AAPCs, either performed as multiple concurrent AAPCs as outlined above or by pipelining multiple AAPCs, the following communication complexity can be derived.

Theorem 2 *A sequence of s all–to–all personalized communications each on d processor dimensions can be accomplished in a time of, at most,*

$$T = \begin{cases} \frac{K}{2} + (s-1)d & s \leq 3 \text{ or } K \leq 8d \\ (s+3)d & 8d \leq K \leq 2(s+1)d \\ \frac{K}{2} + 2d & K \geq 2(s+1)d \end{cases}$$

time steps with all–port communication.

The first case above corresponds to pipelining of AAPCs; the other two cases apply to multiple, concurrent AAPCs. For a succession of AAPCs of different orders, algorithms are presented in [13].

5 Minimizing communications overhead

The scheduling algorithms above attempt to maximize the utilization of the communications bandwidth, i.e., the algorithms strive to communicate on all channels in every exchange step. No attention was paid to a possible overhead in communicating elements. In many message-passing communication libraries, the overhead associated with each message is often substantial. It is of interest to organize the element transfers into block transfers, with each such block being subject to one communications overhead. Below we present a blocking procedure applicable to any algorithm. The procedure yields the minimal number of block transfers, and the minimal block size for this number of block transfers. Applying the blocking procedure to Algorithm 4 yields both optimum span, optimum communication channel utilization, and a minimum number of block transfers in sequence.

We construct the blocks in the form of a table of local memory addresses where all entries in a row can be communicated as one block. Thus, an address can only appear once in a row. A block cannot be

Comm. step	Group 1	Group 2	Group 3	Group 4
0	$\{C(11111), R(01111)\}$	$\{R(00011)\}$	$\{R(00101)\}$	$\{R(00001)\}$
1	$\{C(11111), R(01111)\}$	$\{R(00011)\}$	$\{R(00101)\}$	
2	$\{C(11111), R(01111)\}$	$\{R(00111)\}$	$\{R(01011)\}$	
3	$\{C(11111), R(01111)\}$	$\{R(00111)\}$	$\{R(01011)\}$	
4	$\{C(11111), R(01111)\}$	$\{R(00111)\}$	$\{R(01011)\}$	

Table 8: The scheduling for block exchanges for $d = 5$.

extended beyond a single row. The number of different rows into which an address is entered is equal to the number of exchange steps required for the address. The minimum number of block transfers is equal to the number of rows. Whether or not the minimum number of block transfers can be realized in an implementation depends on the sizes of communications buffers. Clearly, it is of interest to minimize the maximum block size.

Before defining the procedure, we give an example. For Algorithm 4, a table with d rows (the minimum) can be constructed as shown in Table 8. $R(i)$ denotes the necklace of addresses derived from rotations of i, and $C(i)$ denotes the complement pair of addresses $\{i, \bar{i}\}$. Thus, $C(11111)$ denotes the addresses (00000) and (11111). A group is a set of addresses that are scheduled together during d exchange steps in a nonblocked algorithm. Group 1 consists of the c complement pairs of cyclic addresses and the associated $(d-c)$-necklace of addresses scheduled together during d exchange steps. The columns Group 2 and Group 3 each contains the addresses of two full necklaces. The necklaces in a column form d complement pairs of addresses scheduled during d exchange steps. The last column contains a single necklace for which a single exchange suffices. The maximum block size is four in this example.

In Algorithm 1, d complement addresses are scheduled together during each of d exchange steps. Each such group of addresses may form one column of our table for blocking of data exchanges. The number of groups and the block size is $\lceil \frac{K}{2d} \rceil$.

Applying our blocking strategy to Algorithm 2 yields a table of addresses with $\frac{K}{2}$ rows, since the span is $\frac{K}{2}$. Fewer blocks yield bandwidth inefficiency for this algorithm.

In general, we partition the local memory addresses $\mathcal{K} = \{0, 1, \ldots, K-1\}$ into disjoint subsets $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_\alpha$, $\cup_{s=1}^{\alpha} \mathcal{K}_s = \mathcal{K}$, $\mathcal{K}_s \cap \mathcal{K}_r = \phi$, $s \neq r$. The addresses belonging to different subsets are scheduled independently, while addresses within a subset are scheduled together. In the example above, the subsets can be defined as follows: $\mathcal{K}_1 = \{C(11111), R(01111)\}$, $\mathcal{K}_2 = \{R(00011)\}$, $\mathcal{K}_3 = \{R(00111)\}$, $\mathcal{K}_4 = \{R(00101)\}$, $\mathcal{K}_5 = \{R(01011)\}$, and $\mathcal{K}_6 = \{R(00001)\}$.

Let T_s be the time required for subset $\mathcal{K}_s$ to complete the data motion. In our example, $T_1 = 5$, $T_2 = T_4 = 2$, $T_3 = T_5 = 3$ and $T_6 = 1$. By creating a table of addresses with $T_{\max} = \max_{1 \leq i \leq \alpha} T_i$ rows, $T_{\max}$ block transfers are required for a maximum block size of $\lceil (T_{\text{sum}}/T_{\max}) \rceil$, where $T_{\text{sum}} = \sum_{i=1}^{\alpha} T_i$. The strategy in assigning partitions to table entries is that a partition $\mathcal{K}_i$ can appear in a row at most once, but must appear in precisely T_s rows.

Note, that if each subset fully uses the communications bandwidth, then the number of element transfers in sequence is the same for the blocked and nonblocked algorithms, i.e., T_{sum}.

Let $|x|$ be the number of 1-bits in x. Clearly, if the schedule for each subset $\mathcal{K}_i$ fully uses all channels then $T_i = \frac{\sum_{\forall x \in \mathcal{K}_i} |x|}{d}$ and $T_{\text{sum}} = \sum_{i=1}^{\alpha} T_i = \frac{K}{2}$. Thus, we have the following lemma.

Group 1	Time	Group 2	Time	Group 3	Time	Group 4	Time
$\{C(11111), R(01111)\}$	1	$\{R(00011)\}$	0	$\{R(00101)\}$	0	$\{R(00001)\}$	0
$\{C(11111), R(01111)\}$	2	$\{R(00011)\}$	1	$\{R(00101)\}$	1		
$\{C(11111), R(01111)\}$	3	$\{R(00111)\}$	2	$\{R(01011)\}$	2		
$\{C(11111), R(01111)\}$	4	$\{R(00111)\}$	3	$\{R(01011)\}$	3		
$\{C(11111), R(01111)\}$	5	$\{R(00111)\}$	4	$\{R(01011)\}$	4		

Table 9: The scheduling for $d = 5$ and $b = 3$.

Lemma 4 *Algorithm 3 can be organized into* $d + \frac{K}{2} \bmod d \leq 2d - 1$ *block exchanges with maximum block size* $\lceil \frac{K}{2(d+\frac{K}{2} \bmod d)} \rceil$*, which is optimal. Algorithm 4 can be organized into* d *block exchanges with maximum block size* $\lceil \frac{K}{2d} \rceil$*, which is also optimal.*

6 Wide channels

If the channel width is a multiple b of the width of a data item, then $b \times d$ addresses can be scheduled concurrently. Determining the optimum channel utilization is related to minimizing the maximum block size in an algorithm that blocks the data transfers.

Table 9 shows a scheduling for $d = 5$ and $b = 3$. The columns headed by "Time" denote the time step during which an address is scheduled for exchange. The same time step appears precisely $b = 3$ times except for the last step. The time step is determined by labeling the table entries row by row and within each row from right to left.

Lemma 5 *If* $\lfloor \frac{K}{2d} \rfloor \geq b$*, then an AAPC of order* d *with channel bandwidth* b *can be completed in* $d\lceil \frac{K}{2db} \rceil$ *time steps, which is optimal.*

Proof: Multiple occurrences of cyclic addresses appears within the same column (group), since $T_i = d$ for all such addresses. The blocking procedure guarantees that multiple occurrences of cyclic addresses appear within blocks of d rows and in the same column (group). Clearly, multiple occurrences in the same group cannot be scheduled during the same time step, since $b \leq \lfloor \frac{K}{2d} \rfloor$. For noncyclic addresses, $T_i < d$. If the same address appears in two adjacent groups, say in row i of group j and row i' of group $j + 1$, then $i' < i$. ∎

When $b \geq \lceil \frac{K}{2d} \rceil$, the upper bound is the same as the lower bound: d.

7 Summary

We have presented four schedules for a single all–to–all personalized communication, three of which are simple to implement. One algorithm (Algorithm 4) requires the optimal number of element exchanges in sequence, $\frac{K}{2}$, with a span of d. Combining Algorithms 3 and 4 for s all–to–all personalized communications in sequence, each on d dimensions, yields

$$T = \begin{cases} \frac{K}{2} + (s-1)d & s \leq 3 \text{ or } K \leq 8d \\ (s+3)d & 8d \leq K \leq 2(s+1)d) \\ \frac{K}{2} + 2d & K \geq 2(s+1)d) \end{cases}$$

time steps.

The algorithms can be organized such that indirect addressing is not required in interprocessor data exchanges by carrying out pre- and post-alignment steps. By combining these alignments with other local permutations, the presented algorithms can perform permutations of the form $(i|j) \leftrightarrow (g(j)|f(i))$ with no increase in the required data motion.

The presented blocking procedure preserves optimality with respect to element transfers for an optimal element-wise schedule. For instance, for Algorithm 4 the number of block transfers is d for a single AAPC and the block size is $\lceil \frac{K}{2d} \rceil$. Applying the same blocking procedure to the scheduling of exchanges when the channel width is a multiple b of the width of a message yields $\lceil \frac{K}{2b} \rceil$ element transfers in sequence, if $\lfloor \frac{K}{2d} \rfloor \geq b$, otherwise d.

Algorithm 1 has been implemented on the Connection Machine model CM-2. The exchanges require 40 μsec per element compared to 62 μsec for the schedule in [2]. The total expense for alignment and realignment is about 0.9 μsec per element (four bytes). Hence, the simplified algorithms presented here may yield a speedup of up to 50% for a single AAPC as well as a considerably reduced pipeline delay for multiple AAPC.

Acknowledgment

The Connection Machine implementation of Algorithm 1 was performed by Michel Jacquemin of Yale University, Department of Computer Science, in a collaborative effort between Thinking Machines Corporation and INRIA, Centre di Sophia-Antipolis. Valuable comments on a draft were given by Roland Sweet of the University of Colorado at Denver. We also gratefully acknowledge the comments and suggestions made by the referees, which helped improve the presentation significantly.

References

[1] D. P. Bertsekas, C. Ozveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal communication algorithms for hypercubes. *J. of Parallel and Distributed Computing*, 11:263–275, 1991.

[2] Alan Edelman. Optimal matrix transposition and bit-reversal on hypercubes: All-to-all personalized communication. *Journal of Parallel and Distributed Computing*, 11(4):328–331, 1991.

[3] Peter M. Flanders. A unified approach to a class of data movements on an array processor. *IEEE Trans. Computers*, 31(9):809–819, September 1982.

[4] Geoffrey Fox, S. Hiranandani, Kenneth Kennedy, Charles Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[5] Ching-Tien Ho and S. Lennart Johnsson. Optimal algorithms for stable dimension permutations on Boolean cubes. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 725–736. ACM, 1988.

[6] Ching-Tien Ho and S. Lennart Johnsson. Stable dimension permutations on Boolean cubes. Technical Report YALEU/DCS/RR-617, Department of Computer Science, Yale University, October 1988.

[7] Ching-Tien Ho and S. Lennart Johnsson. Spanning balanced trees in Boolean cubes. *SIAM Journal on Sci. Stat. Comp*, 10(4):607–630, July 1989.

[8] D. Hoey and Charles E. Leiserson. A layout for the shuffle-exchange network. In *1980 International Conference on Parallel Processing*. IEEE Computer Society, 1980.

[9] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.

[10] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on Boolean n-cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[11] S. Lennart Johnsson and Ching-Tien Ho. Shuffle permutations on Boolean cubes. Technical Report YALEU/DCS/RR-653, Department of Computer Science, Yale University, October 1988.

[12] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[13] S. Lennart Johnsson and Ching-Tien Ho. Generalized shuffle permutations on Boolean cubes. *J. Parallel and Distributed Computing*, 16(1):1–14, 1992.

[14] S. Lennart Johnsson and Ching-Tien Ho. Boolean cube emulation of butterfly networks encoded by Gray code. *Journal of Parallel and Distributed Computing*, 20(3):261–279, 1994. Department of Computer Science, Yale University, Technical Report, YALEU/DCS/RR-764, February, 1990.

[15] F. Tom Leighton. *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*. MIT Press, Cambridge, MA, 1983.

[16] David Nassimi and Sartaj Sahni. An optimal routing algorithm for mesh-connected parallel computers. *JACM*, 27(1):6–29, January 1980.

[17] David Nassimi and Sartaj Sahni. Optimal BPC permutations on a cube connected SIMD computer. *IEEE Trans. Computers*, C-31(4):338–341, April 1982.

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

[19] Quentin F. Stout and Bruce Wagar. Passing messages in link-bound hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[20] Thinking Machines Corp. *CM–200 Technical Summary*, 1991.

[21] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran – A language specification version 1.1. Technical report, ICASE, Interim Report 21, March 1992.