U.S. DEPARTMENT OF COMMERCE • National Institute of Standards and Technology

NISTIR 88-4001

National Computer Systems Laboratory

Wavefront Matrix Multiplication on a Distributed-Memory Multiprocessor

James Nechvatal

U. S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Systems Laboratory
Advanced Systems Division
Gaithersburg, MD 20899

January 1989

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

Partially sponsored by the
Defense Advanced Research Projects Agency,
1400 Wilson Boulevard,
Arlington, VA 22209.

WAVEFRONT MATRIX MULTIPLICATION ON A DISTRIBUTED-MEMORY MULTIPROCESSOR

James Nechvatal

Advanced Systems Division
National Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Preparation of this report was sponsored in part by the
Defense Advanced Research Agency
1400 Wilson Blvd.
Arlington, Va 22209

U.S. Department of Commerce, C. William Verity, Jr., Secretary

Ernest Ambler, Acting Undersecretary for Technology

National Institute of Standards and Technology, Raymond G. Kammer, Acting Director

January 1989

ABSTRACT

We consider the problem of efficiently multiplying matrices on distributed-memory, message-passing computers. Block decomposition and wavefront computing are employed to yield a communication-efficient solution. We also explore interconnections between distributed data structures, physical networks of nodes, and virtual networks of nodes and processes. The notion of wavefront computing is extended to include pipelining of data between nodes of networks of processes as well as physical networks of nodes. Algorithms are developed in layers to facilitate porting between topologies and programming environments; we also show how different topologies can be employed in a single application. A mathematical characterization of data-routing for efficient matrix multiplication on distributed-memory machines is developed, which exhibits a wavefront version of the Dekel/Nassimi/Sahni algorithm as a special case. We also discuss the notion of granularity in this context and use it to distinguish between algorithms on grounds of communication complexity.

KEY WORDS

distributed; matrix; memory; multiplication; multiprocessor; wavefront

TABLE OF CONTENTS

Page

1. Introduction.

We consider here the problem of efficiently computing C = A * B for M x M matrices A and B on a multiprocesssor with no shared memory. We assume there are s^2 nodes available, each with processor and private memory. Communication between nodes is by packet-switched message-passing, via direct links between nodes. Nodes operate asynchronously. In the algorithms we develop all nodes will execute the same program, with masking of instructions. Hence the full power of MIMD computing is not used, although typically it is available in such machines. An important example of such machines is the hypercube [21].

The application we consider, matrix multiplication, serves as a focal point for a treatment of a number of topics related to message-passing environments, including:

1. Minimization of communication cost.
2. Distributed data structures.
3. Wavefront computing.
4. Networks of processes.
5. Topology of networks.
6. Topology of algorithms.
7. Changing topologies.
8. Modular software.
9. Characterization of relevant algorithms.

Minimization of communication cost is critical for fine-grained computing in particular; here this occurs for some algorithms when M/s is fairly small, and when M/s^2 is small for others. We will return to this in Section 13, where we also observe the effects of different architectures such as the Mark II [24] and the Intel iPSC [5].

Distributed data structures play an important role in minimizing communication, as noted in [11]. They are discussed in the context of hypercubes in, e.g., [12], [14], [20]. In [20] a scheme is developed to permit distributed-memory machines to simulate shared memory. In Section 7 we note that this would simplify some of the discussion here, although we do not incorporate it. In [12] the term areal decomposition is used for the block scheme we employ here. In [3] this is shown to be optimal for the algorithm given there for hypercubes. Block decomposition is also readily adapted to the Dekel/Nassimi/Sahni algorithm [1] which we adopt here.

Our implementation of matrix multiplication is via wavefront computing ([9], [10]). This involves asynchronous pipelining of data through a processor array. It employs the notion of dataflow to modify the systolic approach ([7],[8]). Thus computation at a node is triggered by receipt of operands; there are no synchronization points. The usage of wavefront computing in the context of hypercubes has been noted previously (e.g. [4]).

We also extend the notion of wavefront computing by permitting data to be pipelined not only through the physical network of nodes, but also through a virtual network of processes. We assume that matrix multiplication is only one node in the latter. This second level of pipelining, also asynchronous, may further reduce communication delay. In particular, if processes represent steps in an iterative algorithm, data may flow continuously through the entire algorithm, avoiding possibly costly synchronization at the end of steps.

We regard the s^2 nodes as forming an s x s array, with rows and columns numbered 0, ... ,s-1. Assume that each of A,B,C is an array[0..M-1,0..M-1] of some unspecified element type. Also assume for simplicity that M = m * s for an integer m. We partition A,B,C into s x s arrays of blocks {A(i,j)}, {B(i,j)}, {C(i,j)}, where a block is an array[0..m-1,0..m-1]. In expressions such as A(i,j), parentheses are used for referencing positions in the s x s array of blocks forming A,B,C, or in the s x s array of nodes. Brackets are used to reference elements of data arrays. The block A(i,j) is defined by

$$A(i,j)[k,n] = A[m*i+k,m*j+n] \qquad (0 \le k < m,\ 0 \le n < m,\ 0 \le i < s,\ 0 \le j < s)$$

and similarly for B(i,j) and C(i,j). The awkward notation on the left above and the overloading of the symbols A,B,C will cause no problem since we will never again refer to individual elements of blocks.

The agents for the distribution of {A(i,j)} and {B(i,j)} will be assumed to be previously loaded processes; i.e. for each i and j, on the node in row i, column j of the node array we assume a process Asource(i,j) outputs A(i,j), and a process Bsource(i,j) outputs B(i,j). We will construct process Multiply(i,j), to be loaded to the same node, which will receive A(i,j) and B(i,j) and output C(i,j). Presumably the latter becomes input for yet another process loaded to the same node, namely Csink(i,j). This situation might arise, for example, in an iterative algorithm calling matrix multiplication as a subroutine. The initial configuration of the {(A(i,j)} and {(B(i,j)}, and the final configuration of the {C(i,j)}, are an important part of Multiply; in fact an appropriate way of viewing such an algorithm is via state transitions of such configurations.

Collectively we refer to {Asource(i,j)} as process array Asource, and similarly for {Bsource(i,j)}, {Multiply(i,j)} and {Csink(i,j)}. Presumably Asource, Bsource, Multiply and Csink form part of a network of process arrays hosted by the underlying s x s node array. The latter is a virtual entity formed from the underlying physical network of nodes, which might be a hypercube, toroidal mesh etc. The structure of the network of process arrays is fixed by the application, and we assume the hardware is not reconfigurable. However, we can alter the configuration of nodes in the virtual s x s node array at will. Thus each process array can be hosted by a private node array, whose structure presumably reflects the topology of the process array in some sense. Thus process arrays can execute on virtual machines.

The {A(i,j)} will flow through rows and the {B(i,j)} through columns of the node array, with computation on a node triggered by the receipt of operands. The process arrays Asource, Bsource and Csink may themselves be wavefront-based; then data can flow asynchronously from Asource and Bsource to Multiply, and from the latter to Csink. Thus the flow of data may be viewed as a three-dimensional wavefront through the network of process arrays and the rows and columns of the node array, as illustrated for three nodes in Fig. 1.

Measured in terms of the number of transfers of m x m blocks between neighboring nodes, the wavefront approach gives a communication cost of O(s) for any physical network of nodes with a toroidal mesh as a subnetwork, assuming an initial configuration of data as described above. This cost estimate is essentially independent of the hardware. In contrast, the algorithm in [3] for hypercubes, which is synchronous and broadcast-based, may accrue a cost of up to O(s * log s) depending on packet size. Even if the hardware is augmented with support for broadcasting through rows or columns of the virtual node array, e.g. by broadcasting to subcubes of a cube, the cost is still O(s). The wavefront approach is much more cost-effective in this event, although the advantages of broadcast are greater for meshes [22]. Hence the wavefront approach challenges, at least with regard to matrix multiplication, the recommendation in [2] that hardware support for broadcasting to subcubes be included as a standard for hypercubes.

The existence of algorithms for hypercubes and meshes with O(s) communication cost can be deduced from the development in [1]. This was noted by Johnsson [6], who, however, specified no implementation. The attainability of O(s) cost is also noted in [16].

In Section 7 we note that algorithms oriented to a certain topology may be ported to another by altering the interface to the network of process arrays. This is an instance of the development of modular software for message-passing machines. It requires considerable caution, especially on a machine such as the iPSC where messages are qualified by a type number rather than the identity of the sender.

In the same spirit, in Section 6 we develop a mathematical characterization of algorithms for matrix multiplication on arrays of nodes, independent of interconnection topology. Thus, for example, we are able to view algorithms for toroidal meshes and hypercubes as special cases of a more general data-routing paradigm.

Finally, we note in passing that the traditional method of evaluating the efficiency of algorithms does not fully apply here. Typically matrix multiplication would be a phase in an overlying algorithm; because of pipelining of data between processes, the start and end of a matrix multiplication phase on different nodes is asynchronous. A process representing a single phase can only be evaluated meaningfully in terms of its functioning as a node in the network of process arrays, which presumably represents the overlying algorithm. More generally, it has become clear that new models need to be developed to describe algorithms on distributed-memory machines; these might, e.g., focus on algorithms as manipulations of distributed data structures. Classical theories which view algorithms as mappings from input to output are clearly inadequate in this setting.

2. Node arrays.

Here we formalize the notion of the virtual node arrays mentioned in the previous section. Let S = {0,1, ... , s-1} and let I be the identity matrix of order s. Throughout the paper the symbols s, S and I will be fixed. We also define three data types:

type Boolean = element of {0,1};

Nodelabel = injective mapping from S^2 to {0,1, ... , s^2-1 };

Permutation = permutation on S;

Letting T^* denote transpose of T, a permutation matrix T on S will refer to an array[0..s-1,0..s-1] of Boolean with $T * T^* = I$. We define a bijection Q from the set of Permutations to the set of permutation matrices on S as follows: given a Permutation σ, define a permutation matrix T on S by T[i,j] = 1 if $j = \sigma(i)$ and 0 otherwise; then set $T = Q(\sigma)$. If $T = Q(\sigma)$ then $T^* = Q(\sigma^{-1})$.

DEFINITION 2.1. An adjacency matrix of degree d is a symmetric array[0..s-1,0..s-1] of Boolean with exactly d ones in each row and column, or equivalently all row and column sums equal to d.

□

DEFINITION 2.2. We represent node arrays by ordered pairs:

type Nodearray = (Nodelabel, adjacency matrix);

In Nodearray (P,E), nodes $P(i_1, j_1)$ and $P(i_2, j_2)$ are adjacent if $i_1 = i_2$ and $E[j_1, j_2] = 1$, or if $j_1 = j_2$ and $E[i_1, i_2] = 1$.

□

A Nodearray (P,E) may be interpreted physically as an s x s array of nodes labeled by P; i.e. P(i,j) is the label of the node in row i, column j. Adjacency means that nodes are joined directly by a serial link; the symmetry of E means the links are bidirectional. If E has degree d, then each node is adjacent to exactly d nodes in each row and column, and no others. If a row or column is viewed as an undirected graph then it is regular of degree d, with E determining its edges.

Two Nodearrays (Pstan,Ehyp) and (Pgc,Ecyc) with s = 4 are illustrated in Fig. 2, along with a permutation matrix T. We note that adjacencies of nodes in (Pstan,Ehyp) and (Pgc,Ecyc) are identical; e.g. node 9 is adja-

cent to nodes 1, 8, 11, 13 in both Nodearrays. This is not a coincidence. In Fig. 2 we note that if Pstan and Pgc are treated as arrays, Pstan = T^* * Pgc * T and Ehyp = T^* * Ecyc * T. In general, if E and $\tilde{E}$ are adjacency matrices and $\tilde{E}[i,j] \leq E[i,j]$ for $i,j \in S$ write $\tilde{E} \subseteq E$.

DEFINITION 2.3. Suppose (P,E) and $(\tilde{P},\tilde{E})$ are Nodearrays, T is a permutation matrix on S, $\tilde{P} = T^* * P * T$, and $\tilde{E} \subseteq T^* * E * T$. Then we write $(\tilde{P},\tilde{E}) \subseteq (P,E)$ with transformation matrix T.

□

For example, if (P,E) and $(P, \tilde{E})$ are Nodearrays and $\tilde{E} \subseteq E$ then $(P, \tilde{E}) \subseteq (P,E)$ with transformation matrix I.

LEMMA 2.1. If E and $\tilde{E}$ are adjacency matrices with $\tilde{E} \subseteq E$ and T is a permutation matrix on S, then $T^* * \tilde{E} * T \subseteq T^* * E * T$.

Proof: immediate from $T[i,j] \geq 0$ for all $i,j \in S$.

□

DEFINITION 2.4. If (P,E) and $(\tilde{P},\tilde{E})$ are Nodearrays, T is a permutation matrix on S, $\tilde{P} = T^* * P * T$, and $\tilde{E} = T^* * E * T$, we say $(\tilde{P},\tilde{E})$ is equivalent to (P,E) with transformation matrix T.

□

LEMMA 2.2. If (P,E) and $(\tilde{P},\tilde{E})$ are Nodearrays then the following are equivalent:

i. $(\tilde{P},\tilde{E})$ is equivalent to (P,E) with transformation matrix T.

ii. (P,E) is equivalent to $(\tilde{P},\tilde{E})$ with transformation matrix T^*.

iii. $(\tilde{P},\tilde{E}) \subseteq (P,E) \subseteq (\tilde{P},\tilde{E})$ with transformation matrices T and T^*, respectively.

Proof: suppose (i) holds. Then $\tilde{P} = T^* * P * T$ and $\tilde{E} = T^* * E * T$. Thus $P = T * \tilde{P} * T^*$ and $E = T * \tilde{E} * T^*$. Hence (ii) holds. Similarly (i) holds if (ii) holds. Suppose (i) and (ii) hold. Since $E \subseteq E$ we have $E \subseteq T * \tilde{E} * T^*$, and similarly $\tilde{E} \subseteq T^* * E * T$. Thus (iii) holds. Finally, if (iii) holds, then $\tilde{E} \subseteq T^* * E * T$ and $E \subseteq T * \tilde{E} * T^*$. By Lemma 2.1, $T * \tilde{E} * T^* \subseteq E$ and $T^* * E * T \subseteq \tilde{E}$. Hence (i) and (ii) hold.

□

Lemma 2.2 induces an equivalence relation on the set of Nodearrays. Now if (P,E) and $(P, \tilde{E})$ are Nodear-

rays and $(P, \tilde{E}) \subseteq (P,E)$, then viewing their rows and columns as graphs, the graphs for $(P, \tilde{E})$ are subgraphs of the corresponding graphs for (P,E). More generally we have

LEMMA 2.3. Suppose (P,E) and $(\tilde{P}, \tilde{E})$ are Nodearrays and $(\tilde{P}, \tilde{E}) \subseteq (P,E)$ with transformation matrix $Q(\sigma)$. Then

i. $P(i,j) = \tilde{P}(\sigma(i),\sigma(j))$ $\quad (i,j \in S)$

ii. $E[i,j] \geq \tilde{E}[\sigma(i),\sigma(j)]$ $\quad (i,j \in S)$

iii. if nodes are adjacent in $(\tilde{P},\tilde{E})$ then they are adjacent in (P,E).

Proof: let $T = Q(\sigma)$. Since $\tilde{P} = T^* * P * T$ we have $P = T * \tilde{P} * T^*$, so

$$P(i,j) = \sum_{k=0}^{s-1} \sum_{n=0}^{s-1} T[i,k] * \tilde{P}(k,n) * T[j,n] \qquad (i,j \in S)$$

Also $T[i,k] = 1$ iff $k = \sigma(i)$, and $T[j,n] = 1$ iff $n = \sigma(j)$. This proves (i); (ii) is similar. Now suppose $\tilde{P}(i_1', j_1')$ and $\tilde{P}(i_2', j_2')$ are adjacent in $(\tilde{P},\tilde{E})$. Let $i_r = \sigma^{-1}(i_r')$ and $j_r = \sigma^{-1}(j_r')$, r = 1,2. Now if $i_1' = i_2'$ and $\tilde{E}[j_1', j_2'] = 1$, then $i_1 = i_2$, and by (ii), $E[j_1, j_2] \geq \tilde{E}[\sigma(j_1),\sigma(j_2)] = \tilde{E}[j_1', j_2'] = 1$. Thus $E[j_1, j_2] = 1$. Hence $P(i_1, j_1)$ and $P(i_2, j_2)$ are adjacent in (P,E). The same is true if $j_1' = j_2'$ and $\tilde{E}[i_1', i_2'] = 1$. By (i), $P(i_r, j_r) = \tilde{P}(i_r', j_r')$; this proves (iii).

□

COROLLARY 2.1. If (P,E) and $(\tilde{P},\tilde{E})$ are Nodearrays and $(\tilde{P},\tilde{E})$ is equivalent to (P,E) with transformation matrix $Q(\sigma)$ then

i. $P(i,j) = \tilde{P}(\sigma(i),\sigma(j))$ $\quad (i,j \in S)$

ii. $E[i,j] = \tilde{E}[\sigma(i),\sigma(j)]$ $\quad (i,j \in S)$

iii. Nodes are adjacent in $(\tilde{P},\tilde{E})$ iff they are adjacent in (P,E).

Proof: (i) is direct from Lemmas 2.2iii and 2.3i. By Lemma 2.2iii again, $(P,E) \subseteq (\tilde{P},\tilde{E})$ with transformation matrix $Q(\sigma^{-1})$. Applying Lemma 2.3ii to σ^{-1} we find

$\tilde{E}[i',j'] \geq E[\sigma^{-1}(i'),\sigma^{-1}(j')] \qquad (i',j' \in S)$

Writing $i' = \sigma(i)$, $j' = \sigma(j)$ in the above gives $\tilde{E}[\sigma(i),\sigma(j)] \geq E[i,j]$. Combined with Lemma 2.3ii applied to σ this gives (ii). Similarly (iii) follows from two applications of Lemma 2.3iii.

□

Equivalent node arrays are representations of rearrangements of the same underlying physical network of nodes. Now a process array procarray is a collection of processes {procarray(i,j)}; a priori these have no connections to Nodearrays.

DEFINITION 2.4. We say process array procarray is hosted by Nodearray (P,E) if for $i,j \in S$, procarray(i,j) is loaded to P(i,j).

□

A process array may prefer to be hosted by a certain Nodearray because of alignment of data. For example, a specification in Section 1 is that Multiply(i,j) receives A(i,j) from Asource(i,j) which is also on P(i,j). This interface occurs with minimal communication cost; the restriction is that process arrays Asource and Multiply must be hosted by the same Nodearray. The same is true for Bsource and Multiply. The alignment of {A(i,j)} and {B(i,j)} upon receipt by Multiply is inherited from the topology of the common Nodearray. Presumably this topology is geared to Asource and Bsource. Multiply may wish to have a private topology; i.e. it may wish to be hosted by its own Nodearray. This may be effected physically by a realignment of the {A(i,j)} and {B(i,j)} using a wavefront equivalent to a transformation matrix. In deference to Csink, Multiply can realign {C(i,j)} before passing them along, using the inverse of the previous transformation matrix. In Section 7 we show how this can be done transparently by inserting filters between communicating process arrays hosted by different Nodearrays.

3. Permutations, adjacency and wavefronts.

The characterization of adjacency in Nodearray (P,E) by the definition of E is local; i.e. it tells us where we can send a block from a node with minimal communication cost. Of greater interest is a characterization of how we can interchange blocks among nodes in a row or column concurrently; this will permit us to construct wavefronts flowing through rows, columns or both. Since we do not want to send two blocks to one node, the basic unit for a step of a wavefront is the permutation. A wavefront is thus represented by a composition of permutations. Now we define a family of generic data types by

type Permarray[t] = array[0..t-1] of Permutation;

Generic refers to the parameter t. This definition is in deference to static languages such as C which require the array bound t to be specified at load time, thereby instantiating the data type.

If π is a Permarray[t] then the components of π are designated as π_0 , ... , π_{t-1} , where π_z is a Permutation. Let π^{-1} be the Permarray[t] defined by

$$(\pi^{-1})_z = (\pi_{t-1-z})^{-1} \qquad (0 \le z < t)$$

For the same π let L^{π} be an associated Permarray[t+1] defined by

$$L^{\pi}_z = \pi^{-1}_0 \circ \ \dots \ \circ \pi^{-1}_{z-1} \qquad (1 \le z \le t)$$

$$L^{\pi}_0 (j) = j \qquad (j \in S)$$

It follows that

$$L^{\pi^{-1}}_t = (L^{\pi}_t)^{-1}$$

Also define the ordinary array $L_{\pi} : \{1, \dots ,t\} \times S \rightarrow S$ by

$$L_{\pi}[z,j] = L^{\pi}_z (j) \qquad (1 \le z \le t, j \in S)$$

We will note later that compositions of the form L^{π}_z describe data flow in wavefronts. We will wish to retain the option of disabling the flow at a node during a step; thus we introduce another family of generic types:

type Mask[t] = array[0..t-1,0..s-1] of Boolean;

A Mask is an ordinary array when s and t are instantiated at load time. Full enabling of nodes is represented by J_t, a fixed Mask[t] defined by $J_t[z,j] = 1$ for $0 \leq z < t$, $j \in S$. If π is a Permarray[t] and χ is a Mask[t] let $W^{\chi,\pi}$ be an associated Permarray[s] defined by

$$W_i^{\chi,\pi} = \pi_0^{-\chi[0,i]} \circ \ldots \circ \pi_{t-1}^{-\chi[t-1,i]} \qquad (i \in S)$$

where for any permutation σ, $\sigma^0(j) = j$. For the same χ and π define ordinary array $W_{\chi,\pi} : S^2 \rightarrow S$ by

$$W_{\chi,\pi}[z,j] = W_z^{\chi,\pi}(j) \qquad (z,j \in S)$$

Now in addition to their later use in wavefronts, we can use Permarrays to characterize adjacency in Nodearrays. First we need the following:

DEFINITION 3.1. Suppose π is a Permarray[t] and for each $i \in S$, $\pi_z(i) \neq \pi_{z'}(i)$ for $z \neq z'$. Then we say π is discordant.

□

The study of discordant permutations in combinatorial and statistical settings (e.g. [18]) predates their appearance here by nearly three centuries [13]. For example, a well-known fact is that the probability that an arbitrary Permarray[2] is discordant is very close to 1/e, e = 2.718..., essentially independent of s.

Discordant permutations are fundamental to minimization of communication cost in data-routing on distributed-memory machines. They represent routings of data among a collection of positions so that data never visits the same position twice (cf. Remark 5.1). Moreover, they provide an orthogonal characterization of adjacency for Nodearrays:

DEFINITION 3.2. Suppose π is a discordant Permarray[t]. With Q as in Section 2, let INC(π) be the array[0..s-1,0..s-1] given by

$$\mathrm{INC}(\pi) = \sum_{z=0}^{t-1} Q(\pi_z)$$

Then we say INC(π) is the incidence matrix of π.

□

LEMMA 3.1. Suppose π is a discordant Permarray[t] and E is the incidence matrix of π. Then

i. E[i,j] is the number of z satisfying $j = \pi_z(i)$.

ii. E[i,j] is the number of z satisfying $i = \pi_z^{-1}(j)$.

iii. E is an array of Boolean.

iv. E has exactly t ones in each row and column.

v. if E is symmetric then E is an adjacency matrix of degree t.

Proof: let $T_z = Q(\pi_z)$ for $0 \le z < t$. Then

$$E[i,j] = \sum_{z=0}^{t-1} T_z[i,j] \qquad (i,j \in S)$$

Now $T_z[i,j] = 1$ iff $j = \pi_z(i)$; this proves (i). Also $j = \pi_z(i)$ iff $i = \pi_z^{-1}(j)$; this proves (ii). Now if $j = \pi_z(i) = \pi_{z'}(i)$, since π is discordant we have $z = z'$. By (i), $E[i,j] \le 1$. Since $E[i,j] \ge 0$, (iii) follows. Also

$$\sum_{j=0}^{s-1} E[i,j] = \sum_{z=0}^{t-1} \sum_{j=0}^{s-1} T_z[i,j] = \sum_{z=0}^{t-1} T_z[i,\pi_z(i)] = \sum_{z=0}^{t-1} 1 = t$$

Similarly

$$\sum_{i=0}^{s-1} E[i,j] = \sum_{z=0}^{s-1} T_z[\pi_z(j),j] = t$$

This proves (iv); (v) is immediate.

□

LEMMA 3.2. Suppose E is an adjacency matrix of degree t. Then there exists π, a discordant Permarray[t], with incidence matrix E.

Proof: we note that E is Boolean and has all row and column sums equal to t. By ([19] p. 57) there exist permutation matrices $T_0, \ldots, T_{t-1}$ on S with

$$E = \sum_{z=0}^{t-1} T_z$$

Define π to be the Permarray[t] with $\pi_z = Q^{-1}(T_z)$ for $0 \le z < t$. Since E is Boolean,

$$E[i,j] = \sum_{z=0}^{t-1} T_z[i,j] \leq 1$$

If $\pi_k(i) = \pi_{k'}(i)$ for some i,k,k' with $k \neq k'$, since $T_z = Q(\pi_z)$ we have

$$E[i,\pi_k(i)] \geq T_k[i,\pi_k(i)] + T_{k'}[i,\pi_{k'}(i)] = 2$$

This is a contradiction; hence π is discordant. Also

$$INC(\pi) = \sum_{z=0}^{t-1} Q(\pi_z) = E$$

□

DEFINITION 3.3. Suppose π is a discordant Permarray[t] with $E = INC(\pi)$. Then we say π is a basis for E.

□

Lemma 3.2 says that bases exist for adjacency matrices; but they are not unique.

LEMMA 3.3. Suppose π is a discordant Permarray[t], (P,E) and $(\tilde{P},\tilde{E})$ are Nodearrays, $(\tilde{P},\tilde{E})$ is equivalent to (P,E) with transformation matrix T, and π is a basis for E. Then

i. in (P,E) the nodes adjacent to P(i,j) are

$$\{P(i,\pi_z(j))\}_{0\leq z<t} \quad \cup \quad \{P(\pi_z(i),j)\}_{0\leq z<t}$$

ii. if $\bar{\pi}$ is the Permarray[t] defined by $\bar{\pi} = Q^{-1}(T^* * Q(\pi_z) * T)$ then $\bar{\pi}$ is a basis for $\tilde{E}$.

Proof: we have $E(j, j') = 1$ iff there exists z with $j' = \pi_z(j)$; (i) follows from definition. For (ii) we have

$$E = \sum_{z=0}^{t-1} Q(\pi_z)$$

$$\tilde{E} = \sum_{z=0}^{t-1} T^* * Q(\pi_z) * T = \sum_{z=0}^{t-1} Q(\bar{\pi}_z)$$

Also $\tilde{E} : S^2 \rightarrow \{0,1\}$. Thus $\bar{\pi}$ is discordant and $\tilde{E} = INC(\bar{\pi})$.

□

Lemma 3.3i shows that the adjacency structure of Nodearray (P,E) is characterized by any basis for E.

Lemma 3.3ii shows that the homomorphic image of a basis is a basis for an equivalent node array. Lemma 3.2 guarantees that a basis for E can always be found. Now we need to interconnect these facts with wavefronts. We noted earlier that the latter may be characterized by compositions of permutations. However, these will only be efficient with regard to communication cost if they are related to basis elements. Specifically, each permutation should be achievable in one parallel step involving only communication between neighboring nodes.

DEFINITION 3.4. Call a Permutation σ realizable in Nodearray (P,E) if for $i \in S$, $E(i,\sigma(i)) = 1$ or $\sigma(i) = i$ (in the latter case i is a fixed point of σ). Equivalently, σ is realizable in (P,E) iff $P(i,j)$ is adjacent or equal to $P(i,\sigma(j))$ and $P(\sigma(i),j)$ for $i,j \in S$.

□

Compositions of realizable permutations represent dataflow in a row or column with communication only between adjacent nodes at each step.

DEFINITION 3.5. If π is a Permarray[t] and (P,E) is a Nodearray, we say π is realizable in (P,E) iff π_z is realizable in (P,E) for $0 \le z < t$.

□

If π is a realizable Permarray[t] and L^{π} is discordant, the latter represents a series of data routings yielding minimal communication cost: t nodes are visited by a datum in t steps, with each step involving only transfers between adjacent nodes. As we note later (specifically, during the alignment phase of matrix multiplication), discordancy is sometimes too strong. Fixed points permit masking in some steps. Other violations of discordancy may be indications of nonoptimality in algorithms.

LEMMA 3.4. Suppose (P,E) is a Nodearray and E has degree t. Then

i. If π is a discordant Permarray[t] with no fixed points, $INC(\pi) = E$ iff π is realizable in (P,E).

ii. If π is a discordant Permarray[t'] with no fixed points and π is realizable in (P,E), we can find $\{\pi_z\}_{t' \le z < t}$ so that the extension π is a discordant Permarray[t] with $INC(\pi) = E$.

Proof: exercise.

□

4. Programming and the network of process arrays.

Here we describe our assumptions concerning processes and the communications subsystem which we assume when writing pseudocode. Also we describe the structure and interfacing of the process arrays Asource, Bsource, Multiply and Csink. More generally we address some issues relevant to writing modular software for message-passing machines such as the iPSC.

We assume that a process definition consists of a collection of definitions for procedures, types and symbolic constants, with one procedure designated as main. We assume that a procedures's referencing environment consists of the definitions which are part of the definition of the process containing it, plus locally defined objects. The default parameter transmission mode will be call by value (i.e. in only). If call by result is used (out only) it will be explicitly indicated by the keyword "out" in a procedure heading.

We use < > to refer to generic or unspecific code; e.g. <Element> might be real, double precision etc. If buf is a pointer, *buf refers to the contents of the area buf points to. If arr is an array, arr = a means assigning a to each element of arr.

We assume that at load time a process is supplied with a user-defined integer identifier, with unique identifiers assigned to processes on a node. Furthermore we assume that all processes in a process array are assigned the same identifier; then we define the identifier of the process array to be this common integer.

If a process array procarray is hosted by Nodearray (P,E), we assume that procarray(i,j) may define functions Row( ) and Col( ) which yield i and j, respectively. Presumably these are synthesized from a system call yielding P(i,j).

We define two generic types:

```
type Block = array[0..m-1,0..m-1] of <Element>;
    Blockpointer = pointer to Block;
```

In the above, <Element> may be real etc. We also define standard procedures

```
procedure Null(x,y: Block;
                    a: Blockpointer);
    begin
    end Null;
```

```
procedure Ccomp(x,y: Block;
                a: Blockpointer);
    begin
        < a = a + x * y >
    end Ccomp;
```

The code for Ccomp is unspecific; it computes the product of the blocks x and y in standard sequential fashion.

Each message sent by a process will consist of one Block. A send by a process is a system call of the form

```
send(type,buffer,identifier,node);
```

where type is a user-defined integer qualifier for messages, buffer is a Blockpointer, identifier is the identifier of the process to which *buffer is to be transmitted, and node is the label of the node on which the latter process is located.

A receive by a process is a system call

```
recv(type,buffer);
```

where type and buffer are as above. This fills *buffer with the first block sent to the process with a matching type. This follows the format of the Intel iPSC; in particular the source of a message is not explicitly identified.

We define a unit block transfer by a process to be the issuing of a send to an adjacent node.

LEMMA 4.1. If π is a Permarray[t] realizable in (P,E) and process array procarray is hosted by (P,E) then procarray(i,j) incurs one unit block transfer when it issues a call of the form

```
send(type,buffer,identifier,P(i,π_z(j));        (0 ≤ z < t)
```

or

```
send(type,buffer,identifier,P(π_z(i),j);        (0 <= z < t)
```

Proof: direct from definition.

□

In the following we assume Nodearray (P_0,E_0) is fixed.

DEFINITION 4.1. We assume Csink is hosted by (P_0,E_0) and has identifier 0. Also assume that each Csink(i,j) includes in its code

```
var CL: Blockpointer;
recv(2,CL);
```

Suppose Csink(i,j) issues no receives of type 2 prior to the above, no sends with identifier 1 and no sends with identifier 0 and type 2.

□

DEFINITION 4.2. We assume Asource is hosted by (P_0,E_0) and has identifier ≥ 2. Assume each Asource(i,j) includes in its code

```
var AL: Blockpointer;
    i,j: integer;
i = Row( );
j = Col( );
send(0,AL,1,P0(i,j));
```

Assume Asource(i,j) includes no other assignments to i or j, no other sends with identifier 1, and no sends with identifier 0 and type 2. Suppose also that when it issues the above send, *AL = A(i,j).

□

DEFINITION 4.3. We assume Bsource is hosted by (P_0,E_0) and has identifier ≥ 2. Assume each Bsource(i,j) includes in its code:

```
var BL: Blockpointer;
    i,j: integer;
i = Row( );
j = Col( );
```

send(1,BL,1,P_0(i,j));

Assume Bsource(i,j) includes no other assignments to i or j, no other sends with identifier 1, and no sends with identifier 0 and type 2. Suppose also that when it issues the above send, *BL = B(i,j).

□

DEFINITION 4.4. Suppose f has specification

procedure f(al,bl: Blockpointer;
cl: out Blockpointer;
p: Nodelabel);

Suppose all sends issued by f have identifier 1. Then write f ∈ PREMUL(s,m). Suppose further that all sends and receives issued by f have a ≤ type < b. Then write Typerange(f) = [a,b).

□

Implicitly we assume that if Typerange(f) = [a,b) then a and b-1 are used as types, so that Typerange is well-defined.

DEFINITION 4.5. Suppose f ∈ PREMUL(s,m), Typerange(f) = [a,b), P is a Nodelabel, and for each i,j ∈ S, P(i,j) has some process proc(i,j) with identifier 1 which makes a call of the form f(AL,BL,CL,P). Suppose at the time of this call on P(i,j) we have *AL = A(i,j) and *BL = B(i,j). Suppose any send with identifier 1, issued by any process, with the exception of sends issued within the single call to f in a proc(i,j), has type < a or ≥ b. Suppose any receive issued by a proc(i,j), with the exception of receives issued within its call to f, has type < a or ≥ b. Then let Out(f;P,i,j) be the value of *CL at the time of exit from the call to f on P(i,j).

□

LEMMA 4.2. In Definition 4.5, the value of Out is independent of the {proc(i,j)}.

Proof: the values of *al, *bl and p received by an f are fixed by the hypothesis of Definition 4.5, and *cl is out only. Hence the data received by an f in the form of parameters does not depend on the calling process. Now proc(i,j) is the unique process on a p(i,j) with identifier 1. Hence all sends in a call to an f are to some proc(i,j). Any receives in a proc(i,j) outside its call to f have type < a or ≥ b, and cannot be matched with a send from an f. Thus all sends from within a call to an f are matched with a receive in a call to an f. Conversely,

sends not issued from within an f, with identifier 1, have type < a or ≥ b, and cannot be matched with a receive within an f. Hence all receives within an f are matched by sends from within an f. Thus a copy of f receives data by message transmission only from copies of f; and all copies receive fixed data via parameter transmission. Hence the value of *cl on exit is independent of the calling process.

□

DEFINITION 4.6. Suppose f ∈ PREMUL(s,m). Define g = NETLAYER(f) by

```
procedure g(p: Nodelabel);
    var al,bl,cl: Blockpointer;
        i,j: integer;
    begin
        i = Row( );
        j = Col( );
        recv(0,al);
        recv(1,bl);
        f(al,bl,cl,p);
        send(2,cl,0,p(i,j))
    end g;
```

□

As their names suggest, PREMUL represents potential kernels for multiply routines and NETLAYER surrounds these with an interface to the network of process arrays. Elements of PREMUL thus communicate among themselves exclusively. The following verifies that the interface to the network functions properly.

LEMMA 4.3. Suppose f ∈ PREMUL(s,m), Typerange(f) = [a,b), a ≥ 3 , and g = NETLAYER(f). Suppose process array procarray is hosted by (P_0,E_0) and has identifier 1. Suppose procarray(i,j) has the form

```
<definitions>
procedure main( );
    var P0 : Nodelabel;
    begin
```

<define P_0>

$g(P_0)$

end main.

Suppose procarray, Asource, Bsource and Csink are the only processes loaded. Then

i. procarray(i,j) receives A(i,j) from Asource(i,j) and B(i,j) from Bsource(i,j).

ii. After Csink(i,j) executes recv(2,CL) we have *CL = Out(f;P_0,i,j).

Proof: f issues no sends of type 0, so neither does g, so neither does procarray. Referring to Definitions 4.1 - 4.3, Bsource issues no sends with identifier 1 and type 0; Csink issues no sends with identifier 1. Referring to the format of Definition 4.6, the recv(0,al) in the call to g in procarray(i,j) can only be matched by the send in Asource(i,j) with identifier 1. Conversely, the latter send cannot be matched with a receive in Asource, Bsource or Csink since these have identifiers $\neq$ 1. Also, f has no receives of type 0. Hence the recv(0,al) in g in procarray(i,j) is matched by the send in Asource(i,j) with identifier 1. Thus the former yields *al = A(i,j). Similarly the recv(1,bl) in g in procarray(i,j) gives *bl = B(i,j). This proves (i).

Now in Definition 4.5 take p = P_0 ; note procarray(i,j) makes a call f(al,bl,cl,p) via its call to g. By (i), at the time of this call we have *al = A(i,j) and *bl = B(i,j). Sends by Asource and Bsource with identifier 1 have types < a (in fact 0 or 1); Csink issues no sends with identifier 1. Also, sends by procarray, outside of f, have type < a (in fact = 2). Hence sends not in a call to f, with identifier 1, have type < a. Receives issued by procarray, outside of f, have type < a (in fact 0 or 1). Hence the conditions of Definition 4.5 are satisfied, and the call to f within the call to g in procarray(i,j) gives

*cl = Out(f;P_0,i,j)

Now Asource, Bsource and Csink issue no sends with identifier 0 and type 2. Also, f issues no sends of type 2. Thus the recv(2,CL) in Csink(i,j) can be matched only by the send in g in procarray(i,j). Conversely, the send in g has identifier 0, and hence cannot be matched with a receive in Asource, Bsource, or procarray. Thus it matches the receive in Csink(i,j). This proves (ii).

□

5. A wavefront.

We develop code in this section for what might be termed a symmetric dual masked wavefront flowing through a Nodearray. Dual means data flows horizontally and vertically, and symmetric means the two flows have similar structures. Masked means that only designated nodes are enabled in a given step of one of the flows. If the mask is set to 1 then the flows are homogeneous, i.e. data flow within a row (respectively column) is the same for all rows (respectively columns). Such a wavefront can be coded as

```
procedure Wave(horbuf,verbuf,accum: Blockpointer;
                p: Nodelabel;
                compute: procedure(hbl,vbl: Block;
                                   acc: Blockpointer);
                base,hostid,steps: integer;
                π : Permarray[steps];
                χ : Mask[steps]);
var i,j,z: integer;
begin
    i = row( );
    j = col( );
    for z = 0 to steps-1 do
        begin
            if χ[z,i] = 1 then
                begin
                    send(z+base,horbuf,hostid,p(i,π_z(j)));
                    recv(z+base,horbuf)
                end;
            if χ[z,j] = 1 then
                begin
                    send(z+steps+base,verbuf,hostid,p(π_z(i),j));
                    recv(z+steps+base,verbuf)
```

```
                end;
            compute(*horbuf,*verbuf,accum)
        end
    end Wave;
```

In the above, compute represents activity at a node when data has arrived; accum stores the result; steps is the number of steps comprising the wave. The flow of data is described by compositions of Permutations, namely the components of π. The use of z+base and z+steps+base as types ensures that messages are received in correct order.

DEFINITION 5.1. Suppose P is a Nodelabel and a process on P(i,j) issues a call of the form

Wave(x,y,accum,P,compute,b,hostid,steps,π,χ);

Then this call defines functions h and v where for $0 \le r <$ steps, h(r;i,j) and v(r;i,j) are the values of *horbuf and *verbuf respectively after r iterations of the loop in Wave (these definitions are strictly local to one call on one node).

□

LEMMA 5.1. Suppose P is a Nodelabel. Suppose that for each $i,j \in S$, P(i,j) has a process proc(i,j) with identifier 1 which includes a declaration

```
var X,Y: Blockpointer;
```

and a call

Wave(X,Y,Accum,P,Compute,b,1,t,π,J_t);

Suppose no sends with identifier 1 and $b \le$ type $< b+2t$ are issued by any process except for sends issued within the above call to Wave by a proc(i,j). Suppose no proc(i,j) issues receives with $b \le$ type $< b+2t$ except within its call to Wave. Then for the call to Wave on P(i,j):

i. $h(r+1;i,j) = h(r;i,\pi_r^{-1}(j))$ $(0 \le r < t)$

ii. $v(r+1;i,j) = v(r;\pi_r^{-1}(i),j)$ $(0 \le r < t)$

iii. $h(r;i,j) = h(0;i,L_r^{\pi}(j))$ $(0 \le r \le t)$

iv. $v(r;i,j) = v(0;L_r^{\pi}(i),j)$ $(0 \le r \le t)$

v. the executions of Compute are equivalent to:

for z = 0 to t-1 do Compute($h(0;i,L_{z+1}^{\pi}(j))$,$v(0;L_{z+1}^{\pi}(i),j)$,Accum);

Proof: after r iterations of the loop in Wave on any node, $0 \le r < t$, we have z = r in the next pass through the loop. In the call to Wave on P(i,j) the first receive is

recv(r+b,horbuf);

It is issued from P(i,j) by proc(i,j) which has identifier 1. Since $b \le r+b < b+t$, by hypothesis it can only be matched by a send from some Wave. Since second sends in Waves have type $\ge b+t$, the matching send must be the first. The latter will be issued from some P(i´,j´) and will have the form

send(z´+b,horbuf,1,$P(i',\pi_{z'}(j'))$);

For the receive and send to match, necessarily z´ = r , i´ = i and $\pi_{z'}(j') = j$; the latter gives $j' = \pi_r^{-1}(j)$. Now proc(i,j) is the unique process on P(i,j) with identifier 1. Also, since $b \le z+b < b+2t$, by hypothesis proc(i,j) cannot receive the above send outside its call to Wave. Hence the above send and receive do match. Now this send occurs after r iterations of Wave on P(i´,j´); by Definition 5.1, the value of *horbuf in the send is h(r;i´,j´). Then upon completion of the first receive in Wave on P(i,j) we have *horbuf = h(r;i,j). Now verbuf points to a different area; hence the second receive in Wave on P(i,j) does not alter the above, which becomes the value of *horbuf on P(i,j) after r+1 iterations of the loop in Wave. This proves (i); (ii) is similar.

Now (iii) is true for r = 0; assume true for some r < t. By (i),

$h(r+1;i,j) = h(0;i,(L_r^{\pi} \circ \pi_r^{-1})(j)) = h(0;i,L_{r+1}^{\pi}(j))$

This proves (iii) by induction; (iv) is similar. Also, the sequence of Computes is given by

for z = 0 to t-1 do Compute(h(z+1;i,j),v(z+1;i,j),Accum);

Combined with (iii) and (iv), (v) is an immediate consequence.

□

LEMMA 5.2. Suppose P is a Nodelabel. Suppose for each i,j ∈ S, P(i,j) has a process proc(i,j) with identifier 1 which includes

var X: Blockpointer;

Wave(X,X,nullaccum,P,Null,b,1,t,π,J_t);

Suppose that no sends with identifier 1 and b ≤ type < b+2t are issued by any process, except by a proc(i,j) within its call to Wave. Suppose no proc(i,j) issues receives with b ≤ type < b+2t except in its call to Wave. Then for the call to Wave on P(i,j) we have h = v and

$$h(t;i,j) = h(0;L_t^{\pi}(i),L_t^{\pi}(j)).$$

Proof: horbuf and verbuf point to the same area, so h = v. Now in the pass through the loop in Wave with z = r, the second receive is to the same area as the first. Given i,j ∈ S let $i' = \pi_r^{-1}(i)$ and $j' = \pi_r^{-1}(j)$. Now on P(i',j'), the first send is

send(r+b,horbuf,1,P(i',j));

In the above, *horbuf = h(r;i',j'). By hypothesis this send must be matched by a receive within the Wave on P(i',j). Since b <= r+b < b+t it must be matched by the first receive. The latter is

recv(r+b,horbuf);

By hypothesis, only the above send can match this receive. This transfers h(r;i',j') to horbuf = verbuf on P(i',j). Now the second send on the latter is

send(r+t+b,verbuf,1,P(i,j));

This must be matched with a receive within the Wave on P(i,j). Since first receives have type < b+t, the above must be matched by the second receive on P(i,j), namely

recv(r+t+b,verbuf);

By hypothesis this receive can only be matched by the above send. This transfers h(r;i′,j′) to horbuf = verbuf on P(i,j). Hence

$$h(r+1;i,j) = h(r;\pi_r^{-1}(i),\pi_r^{-1}(j)) \qquad (0 \le r < t)$$

As in Lemma 5.1, it follows by induction that

$$h(r;i,j) = h(0;L_r^{\pi}(i),L_r^{\pi}(j)) \qquad (0 \le r \le t)$$

The result follows.

□

COROLLARY 5.1. With the hypothesis of Lemma 5.2, the action of Wave on P(i,j) is equivalent to

send(b,X,1,P(L_t^{π}(i),L_t^{π}(j));

Proof: immediate from Lemma 5.2.

□

LEMMA 5.3. Suppose P is a Nodelabel. Suppose for each i,j ∈ S, P(i,j) has a process proc(i,j) with identifier 1 which contains

var X,Y: Blockpointer;

Wave(X,Y,nullaccum,P,Null,b,1,t,π,χ);

Suppose no sends with identifier 1 and b ≤ type < b+2t are issued by any process, except by a proc(i,j) within its call to Wave. Suppose no proc(i,j) issues receives with b ≤ type < b+2t except in Wave. Then for the call to Wave on P(i,j) we have

$$h(t;i,j) = h(0;i,W_{\chi,\pi}[i,j])$$

$$v(t;i,j) = v(0;W_{\chi,\pi}[j,i],j)$$

Proof: similar to Lemma 5.1. After r iterations of the loop we take into account whether P(i,j) is enabled horizontally or vertically. This yields

$$h(r+1;i,j) = h(r;i,\pi_r^{-\chi[r,i]}(j))$$

and similarly for v(r+1;i,j). Analogues of the results in Lemma 5.1 are immediate.

□

COROLLARY 5.2. With the hypothesis of Lemma 5.3, the action of Wave on P(i,j) is equivalent to

send(b,X,1,P(i,$W_{\chi,\pi}$[i,j]));

send(b+1,Y,Hostid,P($W_{\chi,\pi}$[j,i],j));

Proof: immediate from Lemma 5.3.

□

REMARK 5.1. If π is realizable, a Wave(...,π,χ) routes all messages explicitly, i.e. sending blocks between adjacent nodes at each step. However, a store-and-forward system such as the iPSC provides virtual circuit service, i.e. implicit routing by the system of messages between non-adjacent nodes. On such a machine the results of Corollaries 5.1 and 5.2 can be used to abrogate the sequence of messages used in Wave in Lemmas 5.2 and 5.3. Implicit routing may reduce overhead, although the cost is the same if measured by unit block transfers. On the other hand, our wavefront routes data with maximum parallelism at each step because of the use of permutations; this holds "conflicts" at nodes to a minimum. The latter is in reference to the buffering necessary when several messages must pass through a node concurrently. In fact, if our wavefront were synchronous (i.e. systolic), system buffering would be unnecessary. To compete with implicit routing, which probably uses paths of minimum length, the wavefront should do the same. In particular L^{π} should be discordant, except for fixed points produced by masking; otherwise some paths will contain cycles. Even with the number of steps (t if π is a Permarray[t]) minimal, however, it may be advantageous to keep both implicit and explicit routing as options in store-and-forward systems.

□

LEMMA 5.4 Suppose π is a Permarray[t] which is realizable in Nodearray (P,E). Suppose a process on P(i,j) makes a call of the form

Wave(x,y,acc,P,compute,b,hostid,t,π,χ);

Then the process will incur at most 2n unit block transfers in executing this call, where n is the maximum

column sum of χ. If $\chi = J_t$ then exactly 2t unit block transfers are incurred.

Proof: direct from Lemma 4.1.

□

6. Matrix multiplication.

We show here that the search for a process array Multiply meeting the specifications of Section 1 can be reduced to the search for certain ordered triples. The main result will show that a suitable class of such triples can be characterized in purely mathematical terms. Examples are given in Sections 8, 9 and 11. Having identified an instance of this class, it is straightforward to turn it into a process array by adding two layers: one to interface with the network of process arrays, and the other to interface with the programming environment. The latter permit development of multiplication modules which can be used with different algorithms and ported between machines, respectively. Furthermore we can change topology: in Section 7 we will show that once we have located a triple for a given Nodearray we can in effect port it to other Nodearrays by adding another outer layer. In Section 12 we give an example.

For s and m as in Section 1, t an integer, P a Nodelabel, π a Permarray[t], ψ a Permarray[s] and χ a Mask[t] define a generic process Mulproc(s,m,t,P,π,ψ,χ) via the code segment

```
< define constant s,m,t: integer;
        type Block,Blockpointer,Nodelabel,Permutation,Boolean,Permarray[ ],Mask[ ];
        constant J_s: Mask[s];
        function Row,Col;
        procedure Wave,Null,Ccomp >
procedure main( );
    var P: Nodelabel;
        AL,BL,CL: Blockpointer;
        i,j: integer;
        π: Permarray[t];
        ψ: Permarray[s];
        χ: Mask[t];
    begin
        < define P,π,ψ,χ >
        i = Row( );
        j = Col( );
```

```
        recv(0,AL);
        recv(1,BL);
        *CL = 0;
        Wave(AL,BL,CL,P,Null,3,1,t,π,χ);
        Wave(AL,BL,CL,P,Ccomp,3+2t,1,s,ψ,J_s);
        send(2,CL,0,P(i,j))
    end main;
```

In the above, Nodelabel, Permutation, Boolean, Permarray[ ] and Mask[ ] are as in Section 2; Block, Block-pointer, Row, Col, Null and Ccomp as in Section 4; and Wave as in Section 5. These and $s,m,t,P,J_s,\pi,\psi,\chi$ must be instantiated to produce a concrete process. The characterization of s,m,t,J_s as symbolic constants external to main( )and P, π,ψ,χ as variables within main( ) is unimportant and merely reflects anticipated programming environments such as Unix. The two waves above represent what Dekel, Nassimi and Sahni [1] refer to as the alignment and shift-multiply phases of a matrix multiplication algorithm. In the following, (P_0,E_0) will be as in Section 4.

DEFINITION 6.1. A Latin square on S is an s x s array whose rows and columns are Permutations.

□

A Latin square on S differs from a discordant Permarray[s] only in that the rows and columns of the latter have a particular labeling.

DEFINITION 6.2. Suppose π is a Permarray[t], ψ is a Permarray[s], and χ is a Mask[t]. Suppose further that L_ψ is a Latin square, $W_{\chi,\pi}$ is symmetric, and

$$W_i^{\chi,\pi} \circ L_k^{\psi} = L_k^{\psi} \circ W_i^{\chi,\pi} \qquad (i \in S;\ 1 \le k \le s)$$

Then write $(\pi,\psi,\chi) \in KER(s)$.

□

The above provides the mathematical characterization we alluded to at the start of the section. In particular, it will permit us to isolate the salient features of the Dekel/Nassimi/Sahni algorithm for hypercubes. The remainder of the section is devoted to showing that the above triples are a suitable class for instantiating

Mulproc, thus providing a unified description of matrix multiplication algorithms on machines with embedded toroidal meshes.

DEFINITION 6.3. Consider the generic procedure definition

```
procedure f(al,bl: Blockpointer;
            cl: out Blockpointer;
            p: Nodelabel);
    var π: Permarray[t];
        ψ: Permarray[s];
        χ: Mask[t];
    begin
        < define π,ψ,χ >
        *cl = 0;
        Wave(al,bl,cl,p,Null,3,1,t,π,ψ);
        Wave(al,bl,cl,p,Ccomp,3+2t,1,s,ψ,J_s)
    end f;
```

If f is as above write $f = F(\pi,\psi,\chi)$. □

LEMMA 6.1. Suppose π is a Permarray[t], ψ is a Permarray[s], and χ is a Mask[t]. If $f = F(\pi,\psi,\chi)$ then

i. $f \in$ PREMUL(s,m).

ii. Typerange(f) = [3,3+2t+2s).

iii. If π and ψ are realizable in Nodearray (P,E) and a process on P(i,j) makes a call of the form

```
f(AL,BL,CL,P);
```

then the process incurs at most 2s+2n unit block transfers in executing this call, where n is the maximum column sum of χ.

Proof: in Definition 6.3, the first Wave has types for sends and receives starting at 3; the second Wave has largest type 3+2t+2s-1. This proves (i) and (ii); (iii) is direct from Lemma 5.4.

□

DEFINITION 6.4. Suppose π is a Permarray[t], ψ is a Permarray[s], and χ is a Mask[t]. Suppose further that Out(F(π,ψ,χ);p,i,j) = C(i,j) for each Nodelabel p and all i,j $\in$ S. Then write (π,ψ,χ) $\in$ MUL(s).

□

THEOREM 6.1. Suppose π is a Permarray[t], ψ is a Permarray[s], χ is a Mask[t], and (π,ψ,χ) $\in$ KER(s). Then (π,ψ,χ) $\in$ MUL(s).

Proof: let f = F(π,ψ,χ). By Lemma 6.1, f $\in$ PREMUL(s,m) and Typerange(f) = [3,3+2t+2s). Suppose P is a Nodelabel and for each i,j $\in$ S, P(i,j) contains a process proc(i,j) with identifier 1 which includes

```
var AL,BL: Blockpointer;
f(AL,BL,CL,P);
```

Suppose *AL = A(i,j) and *BL = B(i,j) on P(i,j) when f is called. Suppose any send with identifier 1 issued by any process, with the exception of sends issued within the single call to f in a proc(i,j), has type < 3 or ≥ 3+2t+2s. Suppose any receive issued by a proc(i,j), with the exception of receives issued within its call to f, has type < 3 or ≥ 3+2t+2s. Now in the call to f on P(i,j), referring to the format of Definition 6.3, we have

Claim 1: on exit from the first Wave in f on P(i,j), *al = A(i, $W_{\chi,\pi}$[i,j]) and *bl = B($W_{\chi,\pi}$[j,i],j).

Claim 2: on exit from the second Wave in f on P(i,j), *cl = C(i,j).

We note that we have satisfied the hypothesis of Definition 4.5, and hence Out(f;P,i,j) is the value of *CL on exit from f on P(i,j), or equivalently the value of *cl on exit from the second Wave. Hence the Theorem follows from Claim 2.

Proof of Claim 1: on entry to the first Wave on P(i,j) we have *al = A(i,j) and *bl = B(i,j). In the notation of Definition 5.1, for this call h(0;i,j) = A(i,j) and v(0;i,j)= B(i,j). On exit from the first Wave *al = h(t;i,j) and *bl = v(t;i,j). Now by hypothesis, no sends with identifier 1 and 3 ≤ type < 3+2t are issued by any process, except by a proc(i,j) in its call to f. Sends and receives issued in the second Wave in f have type ≥ 3+2t; hence no

sends with identifier 1 and 3 ≤ type < 3+2t are issued outside the first Wave in f. Also by hypothesis, any receive issued by proc(i,j) outside of its call to f has type < 3 or ≥ 3+2t. Hence no proc(i,j) issues receives with 3 ≤ type < 3+2t except in the first Wave in its f. Thus the conditions of Lemma 5.3 are satisfied for the first Wave in f on P(i,j), and we have on exit from the first Wave

$$*al = h(t;i,j) = h(0;i, W_{\chi,\pi}[i,j])$$

$$*bl = v(t;i,j) = v(0; W_{\chi,\pi}[j,i],j)$$

Claim 1 follows.

Proof of Claim 2: on entry to the second Wave in f on P(i,j), the values of *al and *bl are as in Claim 1. Now in the notation of Definition 5.1 applied to the second Wave in f,

$$h(0;i,j) = A(i, W_{\chi,\pi}[i,j])$$

$$v(0;i,j) = B(W_{\chi,\pi}[j,i],j)$$

By hypothesis, no sends with identifier 1 and 3+2t ≤ type < 3+2t+2s are issued by any process except for sends by a proc(i,j) in its f. Sends and receives in the first Wave in f have type < 3+2t. Hence no sends with identifier 1 and 3+2t ≤ type < 3+2t+2s are issued outside the second Wave in f in a proc(i,j). Also by hypothesis, no proc(i,j) issues receives with 3+2t ≤ type < 3+2t+2s except in its f. Hence no proc(i,j) issues receives with 3+2t ≤ type < 3+2t+2s except in the second Wave in f. Thus Lemma 5.1 applies to the second Wave, and the executions of Ccomp are equivalent to:

$$\text{for } k = 0 \text{ to } s\text{-}1 \text{ do } Ccomp(h(0;i, L^{\Psi}_{k+1}(j)), v(0; L^{\Psi}_{k+1}(i),j), *cl);$$

or equivalently:

$$\text{for } k = 0 \text{ to } s\text{-}1 \text{ do } Ccomp(A(i, W_{\chi,\pi}[i, L^{\Psi}_{k+1}(j)]), B(W_{\chi,\pi}[j, L^{\Psi}_{k+1}(i)],j), *cl);$$

Taking into account *cl = 0 at the start of f we find that on exit from the second Wave in f on P(i,j),

$$*cl = \sum_{k=0}^{s-1} A(i, W_{\chi,\pi}[i, L^{\Psi}_{k+1}(j)]) * B(W_{\chi,\pi}[j, L^{\Psi}_{k+1}(i)],j)$$

$$= \sum_{k=1}^{s} A(i, W_{\chi,\pi}[i, L^{\Psi}_{k}(j)]) * B(W_{\chi,\pi}[j, L^{\Psi}_{k}(i)],j)$$

$$= \sum_{k=1}^{s} A(i,(W_i^{\chi,\pi} \circ L_k^{\psi})(j)) * B((W_j^{\chi,\pi} \circ L_k^{\psi})(i),j)$$

$$= \sum_{k=1}^{s} A(i,(L_k^{\psi} \circ W_i^{\chi,\pi})(j)) * B((L_k^{\psi} \circ W_j^{\chi,\pi})(i),j)$$

$$= \sum_{k=1}^{s} A(i, L_k^{\psi}(W_{\chi,\pi}[i,j])) * B(L_k^{\psi}(W_{\chi,\pi}[j,i]),j)$$

$$= \sum_{k=1}^{s} A(i, L_k^{\psi}(W_{\chi,\pi}[i,j])) * B(L_k^{\psi}(W_{\chi,\pi}[i,j]),j)$$

Fix i and j for the moment and define (only for this claim)

$$\xi(k) = L_k^{\psi}(W_{\chi,\pi}[i,j]) = L_{\psi}(k, W_{\chi,\pi}[i,j]) \qquad (1 \le k \le s)$$

Then

$$*cl = \sum_{k=1}^{s} A(i,\xi(k)) * B(\xi(k),j)$$

Now L_{ψ} is a Latin square on S; hence its columns are Permutations. Thus ξ is a Permutation. In the last sum above we can write

$$k = \xi^{-1}(k') \qquad (0 \le k' < s)$$

yielding

$$*cl = \sum_{k=0}^{s-1} A(i,k') * B(k',j) = C(i,j)$$

Now i and j are arbitrary; Claim 2 follows.

□

DEFINITION 6.5. Suppose f ∈ PREMUL(s,m). Let g = NETLAYER(f). For i,j ∈ S define Mul(i,j) by

< define type Block,Blockpointer,Nodelabel,Permutation, Boolean, Permarray[ ],Mask[ ];

constant s,t: integer;

J_s: Mask[s];

```
            function Row,Col;
            procedure g,Wave,Null,Ccomp >
        procedure main( );
            var P0 : Nodelabel;
            begin
                < define P0 >
                g(P0 )
            end main;
```

This defines process array Mul. Suppose Mul is hosted by (P_0, E_0) and has identifier 1. Then write Mul = PROGLAYER(f).

□

DEFINITION 6.6. Suppose $(\pi,\psi,\chi) \in$ KER(s) and π and ψ are realizable in Nodearray (P,E). Then write $(\pi,\psi,\chi) \in$ KER(s;P,E).

□

THEOREM 6.2. Suppose t is a positive integer, π is a Permarray[t], ψ is a Permarray[s], χ is a Mask[t], and $(\pi,\psi,\chi) \in$ KER(s;P_0,E_0). For each i,j $\in$ S let Multiply(i,j) = Mulproc(s,m,t,P_0, π,ψ,χ). Then if Asource, Bsource, Multiply, and Csink are the only processes loaded,

i. Multiply(i,j), Asource(i,j), Bsource(i,j) and Csink(i,j) are all loaded to P_0(i,j).

ii. Multiply(i,j) receives A(i,j) and B(i,j) from Asource(i,j) and Bsource(i,j) respectively.

iii. Multiply(i,j) sends C(i,j) to Csink(i,j).

iv. Multiply(i,j) incurs at most 2s+2n unit block transfers, where n = maximum column sum of χ.

Proof: let f = F(π,ψ,χ). Then by Lemma 6.1, f $\in$ PREMUL(s,m), and Typerange(f) = [3,b) where b $\geq$ 3 . Now (i) is direct from Definitions 4.1 - 4.3 and 6.5. In Lemma 4.3 take procarray = Multiply; (ii) is immediate from Lemma 4.3i. Also, since $(\pi,\psi,\chi) \in$ MUL(s), Lemma 4.3ii shows that after Csink(i,j) executes its receive, *CL = Out(f;P_0,i,j) = C(i,j). This proves (iii). Referring to the formats of Definitions 4.6 and 6.5, the call to g(P_0) involves only one send outside the call to f within g, namely

send(2,cl,0,P_0(i,j));

Since the process issuing this send, namely Multiply(i,j), is on P_0 (i,j), this send involves no unit block transfers by our definition. Hence the only unit block transfers come from the call to f in g, and (iv) follows from Lemma 6.1iii.

□

7. Changing topologies.

In Section 4 we noted that process arrays Asource, Bsource, Multiply and Csink should all be hosted by the same Nodearray (P_0, E_0). This minimizes communication cost during interface of processes since no internode communication is involved in passing blocks of A,B,C. However, it forces Multiply to employ the same topology as Asource, Bsource and Csink. A separate topology, represented by an alternative Nodearray (P_1, E_1), may be desirable for the actual multiplication A*B. We can solve this problem by having each Multiply(i,j) include a copy of Wave which acts as a filtering mechanism; this is illustrated in Fig. 3.

The use of $Wave^{-1}$ in Fig. 3 is purely symbolic; i.e. symbolically we have

$$Multiply = Wave^{-1} \circ \tilde{f} \circ Wave$$

Multiply acts as a "block server" for $\tilde{f}$, providing the latter with data configured properly for its topology. The $\tilde{f}$ which appears above is the kernel of the corresponding process array for (P_1, E_1).

DEFINITION 7.1. Suppose $\tilde{f} \in$ PREMUL(s,m), Typerange($\tilde{f}$) = [a,b), P is a Nodelabel, and γ is a Permarray[q]. Then define $f = R(\tilde{f}; \tilde{P}, \gamma)$ by

```
procedure f(al,bl: Blockpointer;
            cl: out Blockpointer;
            P: Nodelabel);
    var nullaccum: Blockpointer;
        b: integer;
    begin
        < define b,P̃,γ >
        Wave(al,al,nullaccum,P,Null,b,1,q,γ^-1,J_q);
        Wave(bl,bl,nullaccum,P,Null,b+2q,1,q,γ^-1,J_q);
        f̃(al,bl,cl,P̃);
        Wave(cl,cl,nullaccum,P,Null,b+4q,1,q,γ,J_q)
    end f;
```

□

LEMMA 7.1. Suppose $\tilde{f} \in$ PREMUL(s,m), Typerange($\tilde{f}$) = [a,b), $\tilde{P}$ is a Nodelabel, and γ is a

Permarray[q]. Let $f = R(\tilde{f}; \tilde{P}, \gamma)$. Then

i. $f \in$ PREMUL(s,m).

ii. Typerange(f) = [a,b+6q).

iii. If γ is realizable in Nodearray (P,E) and a process on P(i,j) makes a call of the form f(AL,BL,CL,P), then the number of unit block transfers incurred by the process in executing this call is 6q plus the number incurred in the call $\tilde{f}$ (AL,BL,CL,$\tilde{P}$).

Proof: all sends by $\tilde{f}$ have a $\leq$ type $<$ b. In f, referring to the format of Definition 7.1, the first Wave has smallest type = b for a send or receive; the third Wave has largest type = b+6q-1 for a send or receive. Also, all sends by $\tilde{f}$ or Waves have identifier 1. This proves (i) and (ii). Now trivially γ^{-1} is realizable in (P,E). Hence each of the three Waves incurs 2q unit block transfers by Lemma 5.4. This proves (iii). □

THEOREM 7.1. Suppose $f \in$ PREMUL(s,m), γ is a Permarray[q], (P,E) and ($\tilde{P}$, $\tilde{E}$) are Nodearrays and ($\tilde{P}$, $\tilde{E}$) is equivalent to (P,E) with transformation matrix $(Q(L_q^{\gamma}))^*$. Then Out(R($\tilde{f}$; $\tilde{P}$,γ);P,i,j) = Out($\tilde{f}$; $\tilde{P}$,i,j).

Proof: abbreviate $\sigma = (L_q^{\gamma})^{-1}$. Let $f = R(\tilde{f}; \tilde{P}, \gamma)$ and Typerange($\tilde{f}$) = [a,b). By Lemma 7.1 we have $f \in$ PREMUL(s,m) and Typerange(f) = [a,b+6q). Suppose for each $i,j \in S$, P(i,j) has some process proc(i,j) with identifier 1 which calls f(AL,BL,CL,P). Suppose that at the time of this call *AL = A(i,j) and *BL = B(i,j). Suppose any send with identifier 1, issued by any process, with the exception of sends by a proc(i,j) within its f, has type $<$ a or $\geq$ b+6q. Suppose any receive issued by a proc(i,j), with the exception of calls within its f, has type $<$ a or $\geq$ b+6q.

Now for the call to f on P(i,j), referring to the format of Definition 7.1, we have

Claim 1: on exit from the second Wave in f on P(i,j), *al = A(σ(i),σ(j)) and *bl = B(σ(i),σ(j)).

Claim 2: on exit from the call to $\tilde{f}$ in f on P(i,j), *cl = Out($\tilde{f}$; $\tilde{P}$,σ(i),σ(j)).

Claim 3: on exit from the third Wave in f on P(i,j), *cl = Out($\tilde{f}$; $\tilde{P}$,i,j).

We note that we have satisfied the hypothesis of Definition 4.5, and hence Out(f;P,i,j) is the value of *CL

on exit from f on P(i,j), or equivalently the value of *cl on exit from the third Wave in f. Hence the Theorem follows from Claim 3.

Proof of Claim 1: on entry to the first Wave in f on P(i,j), *al = A(i,j). In Definition 5.1 applied to the first Wave we have h(0;i,j) = A(i,j). On exit from this call *al = h(q;i,j). By hypothesis, any send outside f with identifier 1 has type < b or ≥ b+2q. Sends and receives in the second and third Waves in f have type ≥ b+2q, and sends and receives in $\tilde{f}$ have type < b, since Typerange($\tilde{f}$) = [a,b). Thus no send with identifier 1, outside the first Wave in f, has b ≤ type < b+2q. By hypothesis no receives issued in proc(i,j) outside of f have b ≤ type < b+2q. The same is true for $\tilde{f}$ and the second and third Waves in f. Hence proc(i,j) issues no receives with b ≤ type < b+2q outside the first Wave in f. Thus the hypothesis of Lemma 5.2 is satisfied for the first Wave with t and π replaced by q and γ^{-1}. Hence on exit from the first Wave in f on P(i,j) we have

$$*al = h(q;i,j) = h(0; L_q^{\gamma^{-1}}(i), L_q^{\gamma^{-1}}(j)) = h(0;\sigma(i),\sigma(j)) = A(\sigma(i),\sigma(j))$$

Similarly, on exit from the second Wave in f on P(i,j) we have *bl = B(σ(i),σ(j)), proving Claim 1.

Proof of Claim 2: on entry to the call to $\tilde{f}$ on P(i,j), we have *al = A(σ(i),σ(j)) and *bl = B(σ(i),σ(j)) by Claim 1. By Corollary 2.1i, P(i,j) = $\tilde{P}$(σ(i),σ(j)). Hence on entry to $\tilde{f}$ on $\tilde{P}$(i,j) = P(σ^{-1}(i),σ^{-1}(j)) we have *al = A(i,j) and *bl = B(i,j).

Sends and receives in the three waves in f have type ≥ b. By hypothesis, sends with identifier 1 outside f have type < a or ≥ b. Hence any send with identifier 1 outside $\tilde{f}$ has type < a or ≥ b. Similarly, receives issued by the proc on $\tilde{P}$(i,j) (as distinguished from proc(i,j)) have type < a or ≥ b, except in $\tilde{f}$. Hence Definition 4.5 applies at this imbedded level with f and P replaced by $\tilde{f}$ and $\tilde{P}$. Hence the value of *cl on $\tilde{P}$(i,j) on exit from $\tilde{f}$ is Out($\tilde{f}$; $\tilde{P}$,i,j). Equivalently, the value of *cl on $\tilde{P}$(σ(i),σ(j)) on exit from $\tilde{f}$ is Out($\tilde{f}$; $\tilde{P}$,σ(i),σ(j)). But by Corollary 2.1i, $\tilde{P}$(σ(i),σ(j)) = P(i,j), proving Claim 2.

Proof of Claim 3: on entry to the third Wave in f on P(i,j), by Claim 2 we have *cl = Out($\tilde{f}$; $\tilde{P}$,σ(i),σ(j)). Invoking Definition 5.1 once again, h(0;i,j) = Out($\tilde{f}$; $\tilde{P}$,σ(i),σ(j)). On exit from the third Wave on P(i,j), *cl = h(q;i,j). A send outside f with identifier 1 has type < a or ≥ b+6q. Sends and receives in the first two Waves have b ≤ type < b+4q. Hence no sends with identifier 1 and b+4q ≤ type < b+6q are issued outside the third Wave. Also, proc(i,j) issues no receives with b+4q ≤ type < b+6q outside the third Wave. Thus Lemma 5.2 applies to the third Wave with t, b and π replaced by q, b+4q and γ. Hence on exit from the third Wave on P(i,j),

$$*cl = h(q;i,j) = h(0;L_q^{\gamma}(i),L_q^{\gamma}(j)) = Out(\tilde{f};\tilde{P},(\sigma \circ L_q^{\gamma})(i),(\sigma \circ L_q^{\gamma})(j)) = Out(\tilde{f};\tilde{P},i,j)$$

This proves Claim 3.

□

COROLLARY 7.1. Suppose π is a Permarray[t], ψ is a Permarray[s], χ is a Mask[t], $(\pi,\psi,\chi) \in MUL(s)$, γ is a Permarray[q], (P_0,E_0) and (P_1,E_1) are Nodearrays, and (P_1,E_1) is equivalent to (P_0,E_0) with transformation matrix $(Q(L_q^{\gamma}))^*$. Let $f = R(F(\pi,\psi,\chi);P_1,\gamma)$. Then $f \in PREMUL(s,m)$, Typerange(f) = [3,b) where $b \geq 3$, and $Out(f;P_0,i,j) = C(i,j)$.

Proof: let $\tilde{f} = F(\pi,\psi,\chi)$. Then by Lemma 6.1, $\tilde{f} \in PREMUL(s,m)$ and Typerange($\tilde{f}$) = [3,b) where $b \geq 3$. Hence $f \in PREMUL(s,m)$ by Lemma 7.1. Since $(\pi,\psi,\chi) \in MUL(s)$, $Out(\tilde{f};P_1,i,j) = C(i,j)$. By Theorem 7.1, $Out(f;P_0,i,j) = C(i,j)$.

□

THEOREM 7.2. Suppose π is a Permarray[t], ψ is a Permarray[s], χ is a Mask[t], (P_0,E_0) is as in Section 4, $(\pi,\psi,\chi) \in KER(s;P_0,E_0)$, (P_1,E_1) is a Nodearray, γ is a Permarray[q] which is realizable in (P_1,E_1), and (P_1,E_1) is equivalent to (P_0,E_0) with transformation matrix $(Q(L_q^{\gamma}))^*$. Let

Multiply = (PROGLAYER $\circ$ R)$(F(\pi,\psi,\chi);P_1,\gamma)$.

Then if Asource, Bsource, Multiply, and Csink are the only processes loaded,

i. Multiply(i,j), Asource(i,j), Bsource(i,j) and Csink(i,j) are all loaded to $P_0(i,j)$.

ii. Multiply(i,j) receives A(i,j) and B(i,j) from Asource(i,j) and Bsource(i,j), respectively.

iii. Multiply(i,j) sends C(i,j) to Csink(i,j).

iv. Multiply(i,j) incurs at most 2s+2n+6q unit block transfers, where n is the maximum column sum of χ.

Proof: let $\tilde{f} = F(\pi,\psi,\chi)$ and $f = R(\tilde{f};P_1,\gamma)$. By Corollary 7.1, $f \in PREMUL(s,m)$. Now (i) is direct from Definitions 4.1 - 4.3 and 6.5. In Lemma 4.3 take procarray = Multiply; (ii) is immediate from Lemma 4.3i. Also by Corollary 7.1 we have $Out(f;P_0,i,j) = C(i,j)$. Lemma 4.3ii gives (iii). Referring to the format of

Definition 6.5, the unit block transfers of Multiply(i,j) are incurred by the execution of $g(P_0)$ on $P_0(i,j)$, where g = NETLAYER(f). As in the proof of Theorem 6.2, the only unit block transfers incurred by the call to g are incurred by its call $f(al,bl,cl,P_0)$ in the format of Definition 4.6. By Lemma 7.1iii, the number of unit block transfers incurred by this call is at most 6q plus the number in $\tilde{f}(al,bl,cl,P_1)$. By Lemma 6.1iii the latter number is at most 2s+2n.

□

We remark that if ε is the unique Permarray[0] with no components, then $(Q(L_0^{\varepsilon}))^* = I$. In Theorem 7.2 we may take $P_1 = P_0$; these are equivalent with transformation matrix I. Also $\tilde{f}$ and $R(\tilde{f};P_0,\varepsilon)$ are indistinguishable. Equating the latter we can view Theorem 6.2 as a special case of Theorem 7.2.

Also, as noted in the Introduction, the machinery of [20] would be useful here. We could then write statements of the form x = p(y), where x is a variable defined on a node and y is a variable defined on another node p. This concept could be extended to procedure names as well. This would simplify and clarify references such as "the proc on P(i,j)." We have not used this machinery for two reasons. The compiler generates communication statements; this would obscure the analysis of communication cost. Also, no existing machine incorporates this machinery.

8. Case study: maximal interconnection.

For comparison purposes we examine the Nodearray (P_0, E_0) with $E_0 = J_s - I$; i.e. each pair of nodes in a row or column is adjacent, the maximal connectivity permitted if we assume that a node is not adjacent to itself. All Permutations are realizable. Let Pstan(i,j) = s*i+j and let P_0 = Pstan. Pstan is illustrated in Fig. 2 for s = 4. If x = q*y+r and $0 \le r < y$ we define x%y = r, x/y = q. Let λ be the cyclic Permutation defined by

$$\lambda(j) = (j-1)\%s \qquad (j \in S)$$

Let π be the Permarray[s] defined by $\pi_z = \lambda^z$ and let $\chi = I$. Then $W^{\chi,\pi}_z = \lambda^{-z}$. Let ψ be the Permarray[s] defined by $\psi_z = \lambda$, $z \in S$. Then $L^{\psi}_z = \lambda^{-z}$, $0 \le z \le s$. Hence

$$W_{\chi,\pi}[z,j] = (j+z)\%s \qquad (z,j \in S)$$

$$L_{\psi}[z,j] = (j+z)\%s \qquad (1 \le z \le s, j \in S)$$

THEOREM 8.1. For a maximally connected s x s Nodearray we can find a process array Multiply meeting the specifications of Section 1, with each Multiply(i,j) incurring 2s+2 unit block transfers.

Proof: without loss of generality take (P_0, E_0) and (π,ψ,χ) as above. Then $W_{\chi,\pi}$ is symmetric and L_{ψ} is a Latin square. Also

$$W^{\chi,\pi}_i \circ L^{\psi}_k = L^{\psi}_k \circ W^{\chi,\pi}_i = \lambda^{-i-k} \qquad (i \in S;\ 1 \le k \le s)$$

Hence $(\pi,\psi,\chi) \in KER(s)$, and trivially $(\pi,\psi,\chi) \in KER(s;P_0,E_0)$. By Theorem 6.2, if each Multiply(i,j) = Mulproc(s,m,s,P_0, π,ψ,χ), Multiply satisfies the conditions of Section 1. Since all column sums of $\chi = I$ are 1, the result follows.

□

9. Case study: toroidal mesh.

Assume $s > 2$; $s = 1$ is trivial and $s = 2$ is covered in the previous section. Let θ be the Permarray[2] with $\theta_0 = \lambda$, $\theta_1 = \lambda^{-1}$, where λ is as in Section 8. Let Ecyc = INC(θ). Ecyc is illustrated for $s = 4$ in Fig. 2. Let P_0 = Pstan as in Section 8, and let E_0 = Ecyc. Then

$$E_0[i,j] = \begin{cases} 1 & (i = (j-1)\%s \text{ or } j = (i-1)\%s) \\ 0 & (\text{otherwise}) \end{cases}$$

Thus each node is adjacent to its four nearest neighbors, with horizontal and vertical connections between the edges of the array; i.e. (P_0, E_0) is a toroidal mesh. Let π be the Permarray[s-1] defined by

$$\pi_z = \begin{cases} \lambda & (z \le s/2 - 1) \\ \lambda^{-1} & (z > s/2 - 1) \end{cases}$$

for $0 \le z \le s-2$. Let χ be the Mask[s-1] defined by

$$\chi[z,k] = \begin{cases} 1 & ((z < k \le s/2) \text{ or } (s/2 < k < 3s/2 - z \text{ and } z \ge s/2)) \\ 0 & (\text{otherwise}) \end{cases}$$

for $0 \le z \le s-2$ and $k \in S$. Then for $k \in S$,

$$W_k^{\chi,\pi} = \pi_0^{-\chi[0,k]} \circ \ldots \circ \pi_{s-2}^{-\chi[s-2,k]}$$

$$= \pi_0^{-\chi[0,k]} \circ \ldots \circ \pi_{s/2-1}^{-\chi[s/2-1,k]} \circ \pi_{s/2}^{-\chi[s/2,k]} \circ \ldots \circ \pi_{s-2}^{-\chi[s-2,k]}$$

Symbolically

$$\log_\lambda W_k^{\chi,\pi} = -\sum_{z=0}^{s/2-1} \chi[z,k] + \sum_{z=s/2}^{s-2} \chi[z,k]$$

$$= \begin{cases} -k & (k <= s/2) \\ s-k & (k > s/2) \end{cases}$$

$$= -k \qquad (k \in S)$$

The last step uses $\lambda^s = \lambda^0$. It follows that

$$W_k^{\chi,\pi} = \lambda^{-k} \qquad (k \in S)$$

We note

$$\sum_{z=0}^{s-2} \chi[z,k] = \begin{cases} k & (k <= s/2) \\ s-k & (k > s/2) \end{cases}$$

Thus the maximum column sum of χ is s/2.

THEOREM 9.1. For an s x s toroidal mesh we can find a process array Multiply meeting the specifications of Section 1, with Multiply(i,j) incurring at most 3s unit block transfers.

Proof: using (P_0, E_0), π and χ as above and taking ψ as in Section 8, we note that $W_{\chi,\pi}$ and L_ψ are as in Section 8. Thus the proof of Theorem 8.1 applies, except that Multiply(i,j) = Mulproc(s,m,s-1,P_0 , π,ψ,χ), and the number of unit block transfers incurred by a Multiply(i,j) is at most $2s + 2(s/2) \leq 3s$. This holds for all $s \geq 1$. □

REMARK 9.1. We have routed data in such a way as to minimize communication cost. However, a simpler algorithm (e.g. [15] p. 126) is obtained by letting $\pi_z = \lambda$ for $0 \leq z \leq s-2$, and $\chi[z,k] = 1$ for $0 \leq z < k$ and 0 for $k \leq z \leq s-2$, $k \in S$. Then the maximum column sum of χ rises to s-1, and the number of unit block transfers becomes 4s-2.

□

10. Butterflies and Gray codes.

Here we record some facts about some classical families of permutations. Given Boolean $x_{d-1}, \ldots, x_0$ let

$$(x_{d-1}, \ldots, x_0)_2 = \sum_{z=0}^{d-1} x_z * 2^z$$

If $x = (x_{d-1}, \ldots, x_0)_2$ we note $x_z = (x/2^z)\%2$. Given nonnegative integers $x^{(1)}, \ldots, x^{(r)}$, suppose

$$x^{(k)} = (x^{(k)}_{d-1} \ldots, x^{(k)}_0)_2 \qquad (1 \le k \le r)$$

Then let

$$EX(x^{(1)}, \ldots, x^{(r)}) = \left(\left(\sum_{k=1}^{r} x^{(k)}_{d-1} \right) \%2, \ldots, \left(\sum_{k=1}^{r} x^{(k)}_0 \right) \%2 \right)_2$$

In particular, EX(x,y) is the exclusive-or of the bit strings representing x and y. We note EX(x) = x = EX(x,0), EX(x,y) = EX(y,x), EX(x,EX(y,z)) = EX(x,y,z), EX(x,x) = 0, and $EX(x,y)/2^z = EX(x/2^z, y/2^z)$. Also, if EX(x,y) = EX(x,w) then y = w. Furthermore, if $x = (x_{d-1}, \ldots, x_0)_2$ and $0 \le z < d$, then

$$EX(x,2^z) = (x_{d-1}, \ldots, x_{z+1}, 1-x_z, x_{z-1}, \ldots, x_0)_2$$

For a fixed z, $EX(x,2^z)$ generates the familiar butterfly data flow, illustrated in Fig. 4 for $x \le 7$ and $z \le 2$. Now EX is not generally additive; however an important exception is

LEMMA 10.1. If $a_0, \ldots, a_{d-1}$ are Boolean then

$$EX(a_0 * 2^0, \ldots, a_{d-1} * 2^{d-1}) = EX\left(\sum_{z=0}^{d-1} a_z * 2^z \right)$$

Proof: trivial. □

Define

$$GC(i) = EX(i,i/2). \qquad (i \ge 0)$$

It is easily verified that GC is injective. The sequence {GC(0), GC(1),...} is the so-called binary reflected Gray code (e.g. [17]). Usually it is developed recursively, but here we have given it a direct definition. To ex-

plore its property of interest, let

$$Z(i) = EX(GC(i),GC(i-1)) = EX(i,i-1,i/2,(i-1)/2) \qquad (i \geq 1)$$

$$Z(0) = 0$$

Then we have

LEMMA 10.2. For $i \geq 1$, Z(i) is the highest power of 2 dividing i.

Proof: let $i = (i_{d-1}, \ldots, i_0)_2$. Suppose $i_r = 0$ for $r < z$, and $i_z = 1$. Then

$$i = (i_{d-1}, i_{d-2}, \ldots, i_{z+1}, 1, 0, 0, \ldots, 0)$$

$$i-1 = (i_{d-1}, i_{d-2}, \ldots, i_{z+1}, 0, 1, 1, \ldots, 1)$$

$$i/2 = (0, i_{d-1}, \ldots, i_{z+2}, i_{z+1}, 1, 0, \ldots, 0)$$

$$(i-1)/2 = (0, i_{d-1}, \ldots, i_{z+2}, i_{z+1}, 0, 1, \ldots, 1)$$

It follows that $Z(i) = 2^z$.

□

COROLLARY 10.1. For $i \geq 0$, the binary representations of GC(i) and GC(i+1) differ in exactly one position.

Proof: immediate from Lemma 10.2.

□

LEMMA 10.3. For $i \geq 0$, EX(Z(0),...,Z(i)) = GC(i).

Proof: this is true for i = 0; assume true for some i. Then

$$EX(Z(0), \ldots, Z(i+1)) = EX(EX(Z(0), \ldots, Z(i)),Z(i+1))$$

$$= EX(GC(i),Z(i+1)) = EX(GC(i),EX(GC(i),GC(i+1)))$$

$$= EX(GC(i),GC(i),GC(i+1)) = EX(GC(i+1))$$

The result follows by induction.

□

LEMMA 10.4. For $i \geq 0$, $GC^{-1}(i) = EX(i,i/2,i/4,i/8,...)$.

Proof: we note $GC(i)/2^j = GC(i/2^j)$. Hence

$$EX(GC(i),GC(i)/2,GC(i)/4, ...) = EX(GC(i),GC(i/2),GC(i/4), ...)$$

$$= EX(i,i/2,i/2/i/4,i/4,i/8, ...) = EX(i) = i$$

The result follows.

□

11. Case study: hypercube.

In this case we take $s = 2^d$. Let $P_0 = Pstan$ as in Section 8. Let π be the discordant Permarray[d] defined by

$$\pi_z(i) = EX(i,2^z) \qquad (i \in S, 0 \le z < d)$$

Let Ehyp = INC(π) and let E_0 = Ehyp. The latter is illustrated in Fig. 2 for s = 4. Let D = {0, ... ,d-1}. By Lemma 3.3, for $i,j \in S$ the nodes adjacent to $P_0(i,j)$ are

$$\{P_0(i,\pi_z(j))\}_{z \in D} \quad \cup \quad \{P_0(\pi_z(i),j)\}_{z \in D}$$

Now k and $\pi_z(k)$ differ only in the z-th digit of their binary representations. Hence the same is true for $P_0(i,j) = s * i + j$ and $P_0(i,\pi_z(j)) = s * i + \pi_z(j)$ for $i,j \in S$. Similarly $s * i + j$ and $s * \pi_z(i) + j$ differ only in one position. Thus in (P_0,E_0), nodes are adjacent iff their binary representations differ in exactly one position. Hence (P_0,E_0) is a hypercube of dimension 2d [21].

We note that if a is Boolean, $\pi_z^a(j) = EX(j,a * 2^z)$, where as usual $\sigma^0(j) = j$ and $\sigma^1(j) = \sigma(j)$ for a permutation σ. More generally we have:

LEMMA 11.1. If $a_0, \ldots, a_r$ are Boolean and $r \in D$,

$$(\pi_0^{a_0} \circ \ldots \circ \pi_r^{a_r})(j) = EX(j, \sum_{z=0}^{r} a_z * 2^z)$$

Proof: the above is true for r = 0; assume true for r-1, $0 < r < d$. Then

$$(\pi_0^{a_0} \circ \ldots \circ \pi_r^{a_r})(j) = (\pi_0^{a_0} \circ \ldots \circ \pi_{r-1}^{a_{r-1}})(\pi_r^{a_r}(j))$$

$$= EX(\pi_r^{a_r}(j), \sum_{z=0}^{r-1} a_z * 2^z) = EX(j,a_r * 2^r, \sum_{z=0}^{r-1} a_z * 2^z) = EX(j, \sum_{z=0}^{r} a_z * 2^z)$$

The last step follows from Lemma 10.1. The result follows by induction. □

We note also $\pi_z^{-1} = \pi_z$. Now let χ be the Mask[d] defined by

$$\chi[z,k] = (k/2^z)\%2 \qquad (z \in D; k \in S)$$

LEMMA 11.2. For $k,j \in S$, $W_k^{\chi,\pi}(j) = EX(j,k)$.

Proof: we have

$$W_k^{\chi,\pi}(j) = (\pi_0^{-\chi[0,k]} \circ \ldots \circ \pi_{d-1}^{-\chi[d-1,k]})(j) = (\pi_0^{\chi[0,k]} \circ \ldots \circ \pi_{d-1}^{\chi[d-1,k]})(j)$$

By Lemma 11.1,

$$W_k^{\chi,\pi} = EX(j, \sum_{z=0}^{d-1} \chi[z,k] * 2^z)$$

The result follows. □

Let GC and Z be as in Section 10 and let ψ be the Permarray[s] defined by

$$\psi_q(j) = EX(j,Z(q)) \qquad (q,j \in S)$$

We note that $\psi_q^{-1} = \psi_q$.

LEMMA 11.3. For $1 \le i \le s$ we have

$$L_q^{\psi} = W_{GC(i-1)}^{\chi,\pi}$$

Proof: we claim

$$(\psi_r \circ \ldots \circ \psi_{i-1})(j) = EX(j,Z(r), \ldots ,Z(i-1)) \qquad (0 \le r < i)$$

This is true for r = i-1; suppose true for some r > 0. Then

$$(\psi_{r-1} \circ \ldots \circ \psi_{i-1})(j) = EX(EX(j,Z(r), \ldots ,Z(i-1)),Z(r-1)) = EX(j,Z(r-1), \ldots ,Z(i-1))$$

Hence the claim follows by induction. Now taking r = 0,

$$(\psi_0 \circ \ldots \circ \psi_{i-1})(j) = (\psi_0^{-1} \circ \ldots \circ \psi_{i-1}^{-1})(j)$$

$$= L_i^{\psi}(j) = EX(j,Z(0), \ldots ,Z(i-1)) = EX(j,GC(i-1))$$

The last step uses Lemma 10.3. The result follows from Lemma 11.2. □

LEMMA 11.4. For i,k ∈ S,

$$W_i^{\chi,\pi} \circ W_k^{\chi,\pi} = W_{EX(i,k)}^{\chi,\pi}$$

Proof: by Lemma 11.2 we have

$$(W_i^{\chi,\pi} \circ W_k^{\chi,\pi})(j) = EX(W_k^{\chi,\pi}(j),i) = EX(EX(k,j),i)$$

$$= EX(i,j,k) = EX(j,EX(i,k)) = W_{EX(i,k)}^{\chi,\pi}(j)$$

□

LEMMA 11.5. For i ∈ S and 1 ≤ k ≤ s we have

$$W_i^{\chi,\pi} \circ L_k^{\psi} = L_k^{\psi} \circ W_i^{\chi,\pi}$$

Proof: immediate from Lemmas 11.3 and 11.4.

□

LEMMA 11.6. For π,ψ,χ as above, (π,ψ,χ) ∈ MUL(s).

Proof: by Lemma 11.2 we have $W_{\chi,\pi}(z,k) = EX(z,k)$. Hence $W_{\chi,\pi}$ is symmetric. Also, by Lemma 11.3, $L_\psi [z,k] = EX(GC(z-1),k)$. Now suppose $L_\psi [z,k] = L_\psi [z',k]$. Then $EX(GC(z-1),k) = EX(GC(z'-1),k)$ and hence $GC(z-1) = GC(z'-1)$. Since GC is injective, $z-1 = z'-1$ and $z = z'$. Thus L_ψ is a Latin square. The result follows from Lemma 11.5.

□

THEOREM 11.1. For a hypercube with s^2 nodes we can find a Nodearray and a process array Multiply meeting the specifications of Section 1 with Multiply(i,j) incurring at most $2(s + \log_2 s)$ unit block transfers.

Proof: let (P_0, E_0) = (Pstan,Ehyp). Let π,ψ,χ be as above and let Multiply(i,j) = Mulproc(s,m,d,P_0, π,ψ,χ). Trivially π is realizable in (P_0, E_0). Also, $\psi_i = \pi_z$ where $z = \log_2 Z(i)$. Hence ψ is realizable in (P_0, E_0). By Lemma 11.6 and Theorem 6.2, Multiply satisfies the conditions of Section 1. Also, the maximum column sum of χ is $d = \log_2 s$. The result follows.

□

As noted in the Introduction, this adapts the Dekel/Nassimi/Sahni algorithm [1] to a wavefront setting.

Furthermore, Theorem 6.1 characterizes the properties of this solution which make it viable; this is somewhat hidden in [1], as indicated by the proof of Theorem 6.1. Extension to the non-square matrix case may be found in [6].

12. Case study: mesh embedded in hypercube.

Here we assume $s = 2^d$, $d > 1$, and that nodes in (Pstan,Ehyp), as in Section 11, are adjacent iff their binary representations differ in exactly one postion. In other words, we again assume the underlying physical network of nodes is a hypercube. However, this time we assume that Asource, Bsource and Csink prefer a different topology than in Section 11. For example they may prefer a subset of a barrel shifter (or PM2I) topology. Assume it is required that (P_0,E_0) has an embedded toroidal mesh as a subnetwork; i.e. with Ecyc as in Section 9 and with the notation of Definition 2.3, according to Lemma 2.3iii we want $(P_0,Ecyc) \subseteq (P_0,E_0)$. Equivalently, $P_0(i,j)$ is to be adjacent to $P_0(i,(j+1)\%2)$ and $P_0((i+1)\%2,j)$. Let $(P_1,E_1) = (Pstan,Ehyp)$ and let π and χ be as in Section 11. Now to employ the machinery of Section 7, we want to find Permutation σ so that (P_1,E_1) is equivalent to (P_0,E_0) with transformation matrix $Q(\sigma)$. The condition $(P_0,Ecyc) \subseteq (P_0,E_0)$ is equivalent to

$$E_0(i,(i+1)\%s) = 1 \qquad (i \in S)$$

By Corollary 2.1ii this is equivalent to

$$E_1(\sigma(i),\sigma((i+1)\%s)) = 1 \qquad (i \in S)$$

Since π is a basis for E_1, the above is equivalent to $\sigma((i+1)\%s) = \pi_z(\sigma(i))$ for some $z = z(i)$. For example, if GC and Z are as in Section 10 and

$$z(i) = \begin{cases} \log_2 Z(i+1) & (0 \le i < s-1) \\ d-1 & (i = s-1) \end{cases}$$

then we find $\sigma(s-1) = 2^{d-1}$ and

$$\sigma(i+1) = EX(\sigma(i),GC(i),GC(i+1)) \qquad (0 \le i < s-1)$$

This suggests that we choose $\sigma = GC$. This is the standard choice; e.g. it is also used in [3]. Let $T = Q(GC)$, $Pgc = T \circ Pstan \circ T^*$ and $Ecyc = T \circ Ehyp \circ T^*$; then we may take $(P_0,E_0) = (Pgc,Ecyc)$. The case $s = 4$ is illustrated in Fig. 2. Also, GC^{-1} is computed in Lemma 10.4. To exploit the machinery of section 7, let γ be the Permarray[d-1] defined by

$$\gamma_z(k) = \pi_z^{\chi[z+1,k]}(k) \qquad (0 \le z < d\text{-}1,\ k \in S)$$

We note

$$\gamma_z(k) = EX(k, \chi[z+1,k] * 2^z)$$

Thus $\gamma_z(k)$ and k have binary representations which do not differ except possibly in the coefficients of 2^z. Hence

$$\chi[r,\gamma_z(k)] = \chi[r,k] \qquad (0 \le r \le d\text{-}1,\ r \ne z)$$

LEMMA 12.1. For $0 \le r < d\text{-}1$ and $k \in S$ we have

$$(\gamma_{d-2} \circ \ldots \circ \gamma_r)(k) = EX(k, \sum_{z=r}^{d-2} \chi[z+1,k] * 2^z)$$

Proof: We claim

$$(\gamma_{d-2} \circ \ldots \circ \gamma_r)(k) = EX(k, \chi[r+1,k] * 2^r, \ldots, \chi[d-1,k] * 2^{d-2})$$

This is true by definition for $r = d\text{-}2$; assume true for some r, $0 < r < d\text{-}1$. Then

$$(\gamma_{d-2} \circ \ldots \circ \gamma_{r-1})(k) = EX(\gamma_{r-1}(k), \chi[r+1, \gamma_{r-1}(k)] * 2^r, \ldots, \chi[d-1, \gamma_{r-1}(k)] * 2^{d-2})$$

$$= EX(k, \chi[r,k] * 2^{r-1}, \ldots, \chi[d-1,k] * 2^{d-2})$$

The claim follows by induction. The Lemma follows from Lemma 10.1. □

LEMMA 12.2. We have $GC = (L_{d-1}^{\gamma})^{-1}$.

Proof: by Lemma 12.1 we have

$$(L_{d-1}^{\gamma})^{-1}(k) = EX(k, \sum_{z=0}^{d-2} \chi[z+1,k] * 2^z) \qquad (k \in S)$$

Now for $k \in S$, we have $\chi[z+1,k] = \chi[z,k/2]$ and $k/2 < 2^{d-1}$. Thus

$$\sum_{z=0}^{d-2} \chi[z+1,k] * 2^z = \sum_{z=0}^{d-2} \chi[z,k/2] * 2^z = k/2$$

The result follows.

□

THEOREM 12.1. For a hypercube with s^2 nodes configured with an embedded toroidal mesh as a subnetwork, we can find a process array Multiply meeting the specifications of Section 1 with Multiply(i,j) incurring at most $2(s - 3 + 4 * \log_2 s)$ unit block transfers.

Proof: we have (P_0,E_0) = (Pgc,Egc) as above. Taking (P_1,E_1) = (Pstan,Ehyp), γ as above and π,ψ,χ as in Section 11, by Lemma 11.6 we have $(\pi,\psi,\chi) \in$ MUL(s). Now by Lemma 12.2, since (P_1,E_1) is equivalent to (P_0,E_0) with transformation matrix $Q(GC) = (Q(GC^{-1}))^*$, Theorem 7.2 applies with q = d-1. The column sums of χ are at most $d = \log_2 s$. The result follows.

□

We remark that Theorem 12.1 exhibits a communication cost $6 * \log_2 s/2$ unit block transfers higher than in Theorem 11.1. This is due to the fact that the blocks of A and B are distributed unfavorably when received from Asource and Bsource, and must be "descrambled" by filtering. In both theorems the communication cost is $2s + O(\log s)$, and hence the cost of filtering is asymptotically negligible. This is why we did not bother to concern ourselves directly with the topology of (P_0,E_0). If the latter is of interest, we can use Lemma 3.3ii to find a basis for E_0.

13. Comparisons.

For comparing results we will employ the unit block transfer for measuring communication complexity; as defined earlier this is the transfer of a block consisting of m^2 matrix elements between neighboring nodes. Results are recorded in Table 1. In the topology column, mesh refers to an ordinary toroidal mesh as defined in Section 9. Mesh in cube refers to a toroidal mesh imbedded in a hypercube, as in Section 12 or [3]; in [6] this is referred to as a Gray code encoding. Cube refers to the arrangement of a hypercube as in Section 11 or [1]; in [6] this is referred to as a binary encoding. Maximal refers to the maximal-connectivity configuration of Section 8.

In the algorithm column of Table 1, FOH refers to Fox/Otto/Hey [3]. The basic algorithm there involves s iterations. Each iteration involves "rolling" blocks of B vertically in parallel at a cost of one unit block transfer, and then broadcasting blocks of A from one position of each row to all other row positions. Each broadcast is a de facto synchronization point. FOH-chain (listed here for later purposes, but omitted in [3]) is the simplest implementation of broadcast, i.e. passing the block serially through the s row positions; each node incurs a delay equivalent to s unit block transfers per iteration. This does not use cube connectivity, and hence is equally applicable to a mesh or mesh in cube. FOH-tree performs the broadcast by embedding a tree in each row of a mesh in cube, requiring at most $\log_2 s$ unit block transfers per node per iteration. FOH-ring refers to a modification of chaining by pipelining packets of a block through a row of a mesh in cube (the split-ring broadcast in [3]). The communication complexity involves n, the number of packets into which a block decomposes. Again this is applicable to a mesh or mesh in cube. FOH-hard refers to broadcast implemented by hardware support; arbitrarily we assume that this involves the same cost as one unit block transfer.

MCBVDV refers to the algorithm in [12], which does not employ block decomposition. If the communication cost is converted, it is equivalent to about s^2 unit block transfers (cf. [16]).

It may be noted that most results in Table 1, and those developed herein in particular, are independent of hardware. As we noted in the Introduction, augmenting the hardware of a cube with a facility for broadcasting to subcubes is strictly a luxury in this context; even if the cost of FOH-hard is less than 2s as listed, it must exceed s (the cost of rolling the B blocks), and hence cannot be much of an improvement over the $2s + \log_2 s$ complexity of the wave adaptation of Dekel/Nassimi/Sahni for cubes. Asymptotically the same is true for the cost of wave for mesh in cube. We also note that increasing the connectivity to the maximum gives only a negligible improvement over the hypercube. In turn, the hypercube does not improve much over the mesh in this context

$(2s + \log_2 s$, versus $3s)$.

It should also be noted that the communication complexity of the algorithm in [3] is given as essentially 2s. This is based on FOH-ring and the assumption that $n >> s^2$. With the latter inequality, pipelining of the n packets of a block through the s row positions is very efficient in implementing broadcast. However, the assumption $n >> s^2$ is specific to the Mark II and its 8-byte packets; in attempting to port FOH-ring to the iPSC and its 1024-byte packets, the inequality $n >> s^2$ may become, e.g., $1 >> 64$. In general the communication complexity of FOH-ring could range from O(s) to $O(s^2)$ depending on m and architecture. In the Introduction we substituted O(s * log s) for an upper bound on this work, based on FOH-tree which is independent of architecture or m.

The impact of communication complexity depends on other costs such as computation, overhead and synchronization, as well as the translation of unit block transfers to actual time cost. Computation time cost is roughly of the form $a * M^3 / s^2$ for some constant a. Time cost of a unit block transfer might be modeled as L + b * n, where L is a constant representing latency, b is a constant and n is the number of packets in a block. For simplicity we will measure packet size in terms of the number p of matrix elements in a packet; then $n = \lceil m^2 / p \rceil$, where $\lceil x \rceil$ denotes the least integer $\geq x$. For example, if elements occupy 8 bytes, the Mark II has p = 1 and the iPSC has p = 128. With these assumptions, execution time may be modeled as

$$time = a*s*m^3 + U(s)*L + U(s)*b*\left\lceil \frac{m^2}{p} \right\rceil + V(m,s)$$

where U(s) is number of unit block transfers and V(m,s) includes overhead and synchronization costs. The latter includes delays when a node cannot compute because it is waiting for receipt of a block. Presumably V(m,s) can be controlled by buffering, use of optimal code, etc.; nonetheless it seems difficult to estimate V with any precision. We will assume $V(m,s) = o(s*m^2)$; this is probably too high for large m, but suffices for analysis.

In the expression for time we note that any term could dominate. For example, V(m,s) may be predominant for small m; $a*s*m^3$ dominates if s is small and m is large; U(s)*L may dominate if m is small, s is large and L >> b; and $U(s)*b*\lceil m^2 / p \rceil$ may dominate if, e.g., $U(s) = s^2$, $m = s^{1/2}$ and s is large. Now if we take U(1) = V(m,1) = 0, we have also

$$efficiency = \frac{1}{1 + \dfrac{B(s)*L + U(s)*b*\left\lceil \frac{m^2}{p} \right\rceil + V(m,s)}{a*s*m^3}}$$

In particular we have

$$efficiency = \frac{1}{1 + O(\frac{1}{m})} \qquad (U(s) = O(s))$$

We also note

$$efficiency = \frac{1}{1 + O(\frac{s}{m})} \qquad (U(s) = O(s^2))$$

In general, if we define granularity as ratio of computation to communication cost,

$$efficiency = \frac{1}{1 + O(\frac{1}{granularity})}$$

We note the central role played by m in these analyses, as opposed to M. This shows that problem size, per se, is a secondary parameter in designing and choosing algorithms for applications such as matrix multiplication. Granularity, e.g. defined by ratio of computation to communication cost, is much more relevant for message-passing machines. We observe that granularity here depends on m if U(s) = O(s), and on m/s if $U(s) = O(s^2)$. Although some of the algorithms in Table 1 fall in between these extremes, it is instructive to note the consequences of this dichotomy. In particular, an O(s) communication complexity, as represented by U(s), has the property of guaranteeing that at least minimal efficiency is attained for any architecture and any values of M and s (of course if s > M only a subset of nodes is used; hence we may restrict $s \leq M$ in practice). To make this more concrete, we introduce:

DEFINITION 13.1. We say that an algorithm is uniformly asymptotically stable with respect to a class of architectures Ξ if for each instance of Ξ there exists a constant e_0 such that efficiency $\geq e_0$ for any problem size and number of nodes.

□

This definition is more of a working nature than theoretical: in practice the number of nodes permitted and problem size will be restricted by the communication and memory subsystems, respectively, for a given architecture. Nonetheless, we conclude that the algorithms in Table 1 with U(s) = O(s), and hence granularity m, are uniformly asymptotically stable with respect to most existing message-passing machines. In fact, as $m \rightarrow \infty$,

efficiency $\rightarrow$ 100%. On the other hand, if $U(s) = O(s^2)$, and hence granularity = m/s, we may have efficency $\rightarrow$ 0 for large s. For example, the Connection Machine [23] has s = 256, and if a matrix element occupies 8 bytes we have $m \leq 8$. Hence $m/s \leq 1/32$, forcing small granularity if $U(s) = O(s^2)$. This is due a priori to memory limitations on the Connection Machine, but more generally we note that execution time is at least $a*M^3/s^2 = a*s^4 (m/s)^3$. Thus on any machine m/s cannot be large for large s. If processor power is considered the situation exacerbates. For example, suppose processors are implemented on single chips whose monetary cost rises linearly with area, and which obey an area-time^2 tradeoff on computation. The total cost of the s^2 processors is then $O(s^2/a^2)$, so that a will in general grow as c*s rather than remain constant for large s. In this case it is more accurate to model execution time as at least $c*M^3/s = c*s^5 (m/s)^3$. Thus if granularity is measured by m/s, small granularity is virtually forced in this context for massively parallel machines. This suggests that porting algorithms such as MCBVDV, FOH-ring and FOH-chain to massively parallel machines is unwise.

In contrast, if $s \leq 4$ the choice of algorithm is probably unimportant. Table 2 gives the results of some runs on a 16-node iPSC, with execution time in seconds. The algorithms tested are seen to be indistinguishable. The reason is the last column, which gives the computation time for the matrix multiplication phase, with communication excised. This shows that high efficiency is attained even for relatively small M, e.g. $M \geq 64$. This is predicted by our model: even if granularity is measured by m/4, for $M \geq 64$ we have $m/4 \geq 4$, so that high granularity is attained quickly.

We note also that FOH-chain is easier to implement than the other three algorithms; hence it seems to be a good choice for this particular machine despite its $O(s^2)$ communication cost. On the other hand, we note that for s = 4, m = 64, communication cost may be as much as one second. Our theory predicts that this cost would rise to about 64 seconds on a similar (but nonexistent) 1024-node machine, compared to a 50-second computation time. The wave version should reduce communication cost to about 5 seconds in this case.

14. Conclusions.

Using matrix multiplication as an example, we have explored various aspects of computing on distributed-memory, message-passing machines. The focus has been environmental rather than algorithmic, except for Section 6. We have noted the importance of the interface between processes, and its connection with distributed data structures. We have seen that wavefront computing is an effective means of implementing algorithms; futhermore, we have extended the notion of wavefront computing to include dataflow between nodes of networks of process arrays. This permits iterative algorithms to be constructed from wavefront-based modules which can be interfaced so that data flows continuously rather than having the end of an iteration serve as a de facto synchronization point. Furthermore, we have seen that interfaces between process arrays can be constructed so as to permit each process array to execute on its own virtual node array. Thus, iterative algorithms can be constructed from routines which are heterogeneous with respect to topology.

There are various ways in which the present work needs to be extended. In the future we plan to give examples of iterative algorithms calling matrix multiplication as a subroutine, thereby giving concrete examples of the usage of the machinery developed here. Also, here and elsewhere the role of distributed data structures has been observed; in fact it has been noted that algorithms on distributed-memory machines could be characterized as mappings between such structures. This notion needs to be developed further, since it is a radical departure from the classical theory of algorithms which has been based on models such as Turing machines.

BIBLIOGRAPHY

1. E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms," SIAM J. Comput., vol. 10, no. 4, pp. 657-673, Nov. 1981, pp. 657-673.

2. G. C. Fox and W. Furmanski, "Communication algorithms for regular convolutions and matrix problems on the hypercube," in Hypercube Multiprocessors 1987, Proc. 2nd Conf. Hypercube Multiprocessors, M. T. Heath, Ed., 1986, pp. 223-238.

3. G. C. Fox, S. W. Otto and A. J. G. Hey, "Matrix algorithms on a hypercube I: matrix multiplication," Parallel Comput., vol. 4, no. 1, pp. 17-31, Feb. 1987.

4. M. T. Heath and C. H. Romine, "Parallel solution of triangular systems on distributed-memory multiprocessors," SIAM J. Sci. Stat. Comput., vol. 9, no. 3, pp. 558-588, May 1988.

5. Intel Corporation, iPSC Overview, 1986.

6. S. L. Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," J. Parallel Distrib. Comput., vol. 4, no. 2, pp. 133-172, Apr. 1987.

7. H. T. Kung, "Why systolic architectures," Computer, vol. 15, no. 1, pp. 37-46, Jan. 1982.

8. H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in Sparse Matrix Proceedings, I. Duff et. al., Eds., 1978, pp. 245-282.

9. S.-Y. Kung, "On supercomputing with systolic/wavefront array processors," Proc. IEEE, vol. 72, no. 7, pp. 867-884, July 1984.

10. S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. B. Rao, "Wavefront array processor: language, architecture, and applications," IEEE Trans. Comput., vol. C-31, No. 11, pp. 1054-1066, Nov. 1982.

11. B. Lint and T. Agerwala, "Communication issues in the design and analysis of parallel algorithms," IEEE Trans. Software Eng., vol. SE-7, no. 2, pp. 174-188, Mar. 1981.

12. O. A. McBryan and E. F. Van de Velde, "Hypercube algorithms and implementations," 2nd SIAM Conf.

Parallel Processing for Scientific Computing, 1985, in SIAM J. Sci. Stat. Comput., vol. 8, no. 2, pp. s227-s287, Mar. 1987.

13. P. R. Montmort, Essai d'Analyse sur les Jeux de Hazard. Paris: 1708.

14. Z. Mu and M. C. Chen, "Communication-efficient distributed data structures on hypercube machines," in Hypercube Multiprocessors 1987, Proc. 2nd Conf. Hypercube Multiprocessors, M. T. Heath, Ed., 1986, pp. 67-77.

15. M. J. Quinn, Designing Efficient algorithms for Parallel Computers. New York, NY: McGraw-Hill, 1987.

16. M. J. Quinn and S. J. Sulsky, "A fast matrix multiplication algorithm for hypercube multicomputers," Presented at the 3rd Conf. Hypercube Concurrent Computers and Applications, Pasadena, CA, Jan. 1988.

17. E. M. Reingold, J. Nievergelt and N. Deo, Combinatorial Algorithms: Theory and Practice. Englewood Cliffs, NJ: Prentice-Hall, 1977.

18. J. Riordan, An Introduction to Combinatorial Analysis. Princeton, NJ: Princeton University Press, 1978.

19. H. Ryser, Combinatorial Mathematics. Carus Mathematical Monographs No. 14, New York: Mathematical Association of America, 1963.

20. L. R. Scott, J. M. Boyle and B. Bagher, "Distributed data structures for scientific computation," in Hypercube Multiprocessors 1987, Proc. 2nd Conf. Hypercube Multiprocessors, M. T. Heath, Ed., 1986, pp. 55-66.

21. C. L. Seitz, "The cosmic cube," Commun. ACM, vol. 28, no. 1, pp. 22-33, Jan. 1985.

22. Q. F. Stout, "Mesh-connected computers with broadcasting," IEEE Trans. Comput., vol. C-32, no. 9, pp. 826-830, Sep. 1983.

23. Thinking Machines Inc., Introduction to Data Level Parallelism. Cambridge, MA: Thinking Machines Technical Report 86.14, April 1986.

24. J. Tuazon, J. Peterson, M. Pniel and D. Liberman, "Caltech/JPL Mark II hypercube concurrent processor," Proc. Int. Conf. Parallel Processing, St. Charles, IL, Aug. 1985, pp. 666-673.

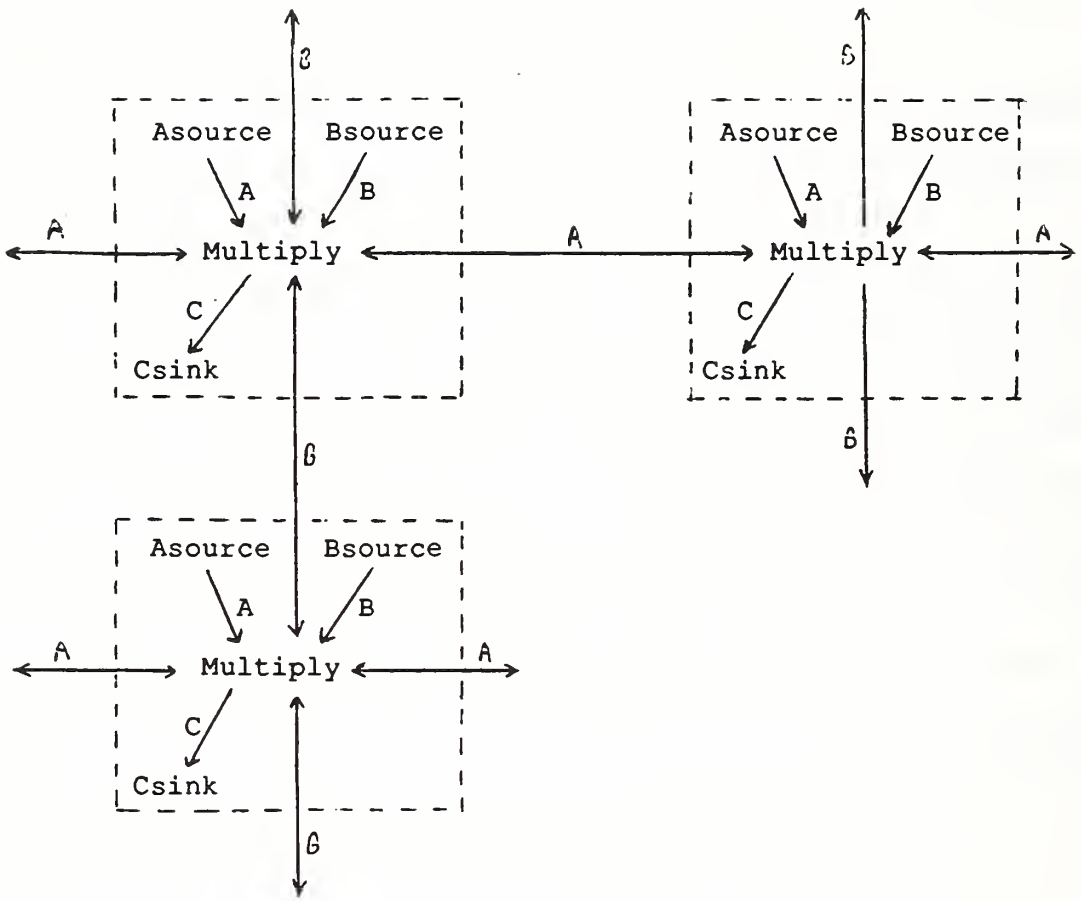
Asource
Bsource
A
B
A
Multiply
A
C
Csink
Asource
Bsource
A
B
A
Multiply
A
C
Csink
Asource
Bsource
A
B
A
Multiply
A
C
Csink

Fig. 1

$$
\text{Pstan} = \begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix}
\qquad
\text{Ehyp} = \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{matrix}
$$

$$
\text{Pgc} = \begin{matrix} 0 & 1 & 3 & 2 \\ 4 & 5 & 7 & 6 \\ 12 & 13 & 15 & 14 \\ 8 & 9 & 11 & 10 \end{matrix}
\qquad
\text{Ecyc} = \begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix}
$$

$$
\text{T} = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix}
$$

Fig. 2

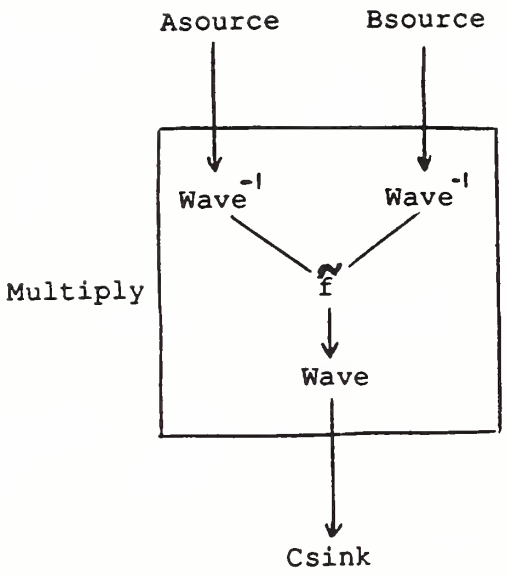

Fig. 3

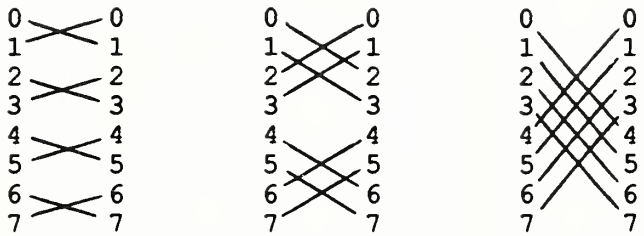
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
EX(x,1)
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
EX(x,2)
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
EX(x,4)

Fig. 4

topology	algorithm	reference	unit block transfers
mesh	FOH-ring	[3]	$\cong (s^2 - 2s)/2n + s$
mesh	FOH-chain	[3]	s^2
mesh	wave-simple	Remark 9.1	$4s - 2$
mesh	wave-optimal	Theorem 9.1	$3s$
mesh in cube	FOH-chain	[3]	s^2
mesh in cube	FOH-tree	[3]	$s * \log_2 s$
mesh in cube	FOH-ring	[3]	$\cong (s^2 - 2s)/2n + s$
mesh in cube	wave	Theorem 12.1	$2s - 6 + 8 * \log_2 s$
mesh in cube	FOH-hard	[3]	$2s$
cube	MCBVDV	[12]	$\cong s^2$
cube	wave	Theorem 11.1	$2s + \log_2 s$
maximal	wave	Theorem 8.1	$2s + 2$

Table 1

s	m	wave	FOH-tree	FOH-ring	FOH-chain	comp
1	4	5	5	5	5	5
1	8	30	35	35	30	30
1	16	210	225	220	225	210
1	32	1595	1640	1615	1645	1590
1	64	12375	12555	12460	12570	12355
2	4	20	15	20	15	10
2	8	70	70	70	70	60
2	16	515	445	445	445	420
2	32	3310	3250	3250	3260	3165
2	64	24850	24990	24965	25015	24645
4	4	65	95	100	40	20
4	8	260	205	195	145	115
4	16	1085	940	945	950	830
4	32	6595	6665	6595	6665	6310
4	64	49830	50345	49985	50250	49210

Table 2

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NISTIR 88-4001	2. Performing Organ. Report No.	3. Publication Date January 1989

4. TITLE AND SUBTITLE

Wavefront Matrix Multiplication on a Distributed-Memory Multiprocessor

5. AUTHOR(S)

James R. Nechvatal

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

We consider the problem of efficiently multiplying matrices on distributed-memory, message-passing computers. Block decomposition and wavefront computing are employed to yield a communication-efficient solution. We also explore interconnections between distributed data structures, physical networks of nodes and virtual networks of nodes and processes. The notion of wavefront computing is extended to include pipelining of data between nodes of networks of processes as well as physical networks of nodes. Algorithms are developed in layers to facilitate porting between topologies and programming environments; we also show how different topologics can be employed in a single application. A mathematical characterization of data-routing for efficient matrix multiplication on distributed-memory machines is developed, which exhibits a wavefront version of the Dekel/Nassimi/Sahni algorithm as a special case. We also discuss the notion of granularity in this context and use it to distinguish between algorithms on grounds of communication complexity.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

distributed; matrix; memory; multiplication; multiprocessor; wavefront

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

70

15. Price

$13.95