Vector Computations on an Orthogonal Memory Access Multiprocessing System

Isaac D. Scherson and Yiming Ma

Department of Electrical and Computer Engineering
University of California
Santa Barbara, Ca. 93106

ABSTRACT

An Orthogonal Memory Access system allows a multiplicity of processors to concurrently access distinct rows or columns of a rectangular array of data elements. The resulting tightly-coupled multi-processing system is feasible with current technology and has even been suggested for VLSI as a "reduced mesh". In this paper we introduce the architecture and concentrate on its application to a number of basic vector and numerical computations. Matrix multiplication, L-U decomposition, polynomial evaluation and solutions to linear systems and partial differential equations, all show a speed-up of O(n) for a n-processor system. The flexibility in the choice of the number of PEs makes the architecture a strong competitor in the world of special-purpose parallel systems. Actually, we prove that the machine exhibits the same performance as any other system with the same number of processors within a factor of 3.

KEYWORDS: Parallel algorithms, Parallel architecture, Numerical analysis, Time complexity, Performance analysis.

I. INTRODUCTION

With the fast development of VLSI technology, various high speed computer architectures have been proposed. Problems that were too large in the past are now solvable. As a result, people try to solve larger problems and are asking for even faster computers. Nowadays, however, it appears that the speed of the electronic components is reaching its physical limits. Improving the hardware is no longer an attractive way to achieve higher speed. For the last two decades, researchers have paid more and more attention to the structure of the computer system so that certain operations can be executed in parallel or at least in an overlapping manner. The results of these efforts have brought forth various commercially available multi-processor systems. In parallel with the growth of parallel computers, parallel algorithms have gone from theoretical games to practical interest.

Perhaps the biggest difference between serial and parallel algorithms is that parallel algorithms are machine dependent. An excellent parallel algorithm for one machine might not apply to another machine at all. Thus, one problem might be neatly solved in one machine but poorly solved in another. Because of this problem oriented nature, most of todays parallel computers are special-purpose or semi-special-purpose. This is especially true for systems composed of a large number of processors.

In this paper, we shall first describe an orthogonal memory access (OMA) machine, which was proposed independently by two groups, Tseng, Hwang and Kumar[1] and the other group led by Isaac Scherson[3]. We shall analyze the communication problem of the machine in some detail. The main result of the analysis is that the OMA is at least as fast as any other multiprocessor system within a factor of 3.

Another issue of this paper is to develop some parallel numerical algorithms for the OMA computer. The serial versions of these algorithms have been thoroughly discussed and refined during the last one hundred years. Many techniques were developed. Some of which were proven to be optimal; while others were attractive for their simplicity or for their superior behavior at some special cases. The choice of the problems and methods to be implemented are based on the following criteria :

1. They are most commonly used.

2. The variety of problems cover the basic techniques of numerical analysis.

As a result of our decision, five topics will be discussed : 1) matrix multiplication, 2) L-U decomposition, 3) triangular systems 4) partial differential equations and 5) polynomial manipulation. They can be found in any reasonably comprehensive numerical analysis book (e.g. [2]). Some restricted or unrestricted parallel implementations are scattered over a number of references (e.g. [4] and [6]). All of the algorithms presented here have linear speedups over the corresponding serial ones. Since our machine has n processors, a linear speedup is all one can expect. Some more numerical algorithms with linear speedups can be found in [1]. Thus, it is not difficult for one to see that the new system serves as a semi-general purpose machine at least in the field of numerical analysis.

II. ANALYSIS OF THE OMA

Due to the nature of numerical vector computations in a parallel processing environment, concurrent access to elements of a rectangular array is often desirable. Simutaneous access to a row or a column or a main diagonal in a two dimensional array are good examples of operations of this type. These occur when we try to compute a dot product, do a Gaussian elimination, transpose a matrix or calculate the trace of a matrix. In ILLIAC IV [18] like machines, such as STARAN [19] and MPP [20], skewing or scrambling of the elements of the array was used to achieve vector accessability. This idea motivates the design of an orthogonal memory access machine (OMA) to facilitate parallel manipulation of rows and columns.

2.1 Brief descriptions of the OMA

An n-processor OMA has n×n memory modules organized in the way depicted in Figure 1. We shall use

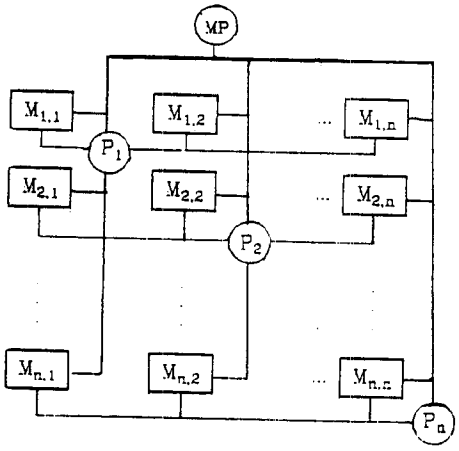


Figure 1: The schematic diagram of the OMA machine.

the symbol M_{ij} for the memory module on the i-th row, j-th column, and P_i for the i-th processor. For our discussion, we shall omit any part that does not concern us, including the I/O portion of the system. There is a master processor for control. It can initiate or mask off any of the processors and provides two memory mapping modes:

RM (Row mapping) : Processor P_i has access to and only to any of the memory modules in the i-th row.

CM (Column mapping) : Processor P_j has access to and only to any of the memory modules in the j-th column.

Given an n by n matrix $Q = [q_{ij}]$ stored in the n×n memory array such that $q_{ij} \varepsilon M_{ij}$. In a RM mode, any column can be accessed simultaneously, that is each P_i can fetch a q_{ix}. Similarly, CM allows parallel access to rows of Q. Furthermore, we note that the diagonal of Q can be concurrently fetched in either memory mapping mode. It is also noted that for a matrix of size larger than n, a fragment of a row or column or diagonal vector can be accessed in parallel.

Each processor P_i is capable of puting an element in the same address of all memory modules in the i-th row when the machine is operating in a RM mode, and conversely, does the same for the i-th column when the machine is operating in a CM mode. Thus each processor can broadcast to any other processor(s) in two memory cycles.

Another optional feature which is easy to implement is the shifting operation. In a RM mode, processor P_i can set up a shifting channel such that the content of the appropriate address in each memory module within the i-th row gets shifted to the next memory module within the same row in one memory cycle. Of course, one can do the same thing for columns when the machine is operating in a CM mode.

A two way switch is attached to each memory module M_{ij} with i not equal j to allow either P_i or P_j to have access to it. For an OMA of size n, there are n horizonal buses. Each bus connects P_i to the n-1 two-way switches of M_{ij} with i not equal j and to M_{ii} directly. Similarly, n vertical buses link the processors and the switches of memory modules in the same column. All the n^2 - n switches are controlled by a single bit, which is set or reset by the master processor.

2.2 Communication analysis

A network **does not have path conflicts** if any pair of nodes can have a dedicated path to communicate if both of them are idle. Let us define the distance between a pair of processors to be the minimum number of steps necessary to move a piece of data from one processor to the other. Assume bidirectional links, so that we don't have to distinguish between the source and destination. The diameter of a network is the maximum distance among all pairs of processors. For example, the diameter of an n×n mesh-connected network without wrap-around is 2n-2. A multistage switch network with logn stages (e.g. Benes network, Shuffle-exchange network, ω-network and cube network) has a diameter of logn. It is obvious that for networks without path conflicts the shorter the diameter, the less communication overhead the system will suffer. It is also true that both the network complexity and the communication overhead increase with the number of processors. Let us look at the OMA. Since every processor can put or fetch data in any of the memory modules within the same row or the same column, the diameter is 2. Let us analyze how the diameter affects the communication complexity. We shall distinguish two types of steps: 1) Computational steps (COMPS) and 2) Communication steps (XFERS), respectively.

Theorem 1 : Let N be the number of COMPS of an unrestricted parallel algorithm A, and let r be the maximum number of operands that any operation in A could have. Then at most (r + 1)N steps are needed for the OMA to execute A.

proof : At the end of **each** step, insert XFERS (or broadcast) if necessary. Each XFERS makes one operand available to every processor. Since each operation needs at most r operands, at most r COMPS will make every processor ready for the next operation. Therefore a total of rN XFERS are needed, in addition to the N COMPS of the unrestricted algorithm. The total execution time for algorithm A in OMA becomes then (r + 1)N.

Theorem 2 : Assume each COMPS is either unary or binary (as conventional). If N is the number of steps for some n-processor system to execute a parallel algorithm A, then at most 3N steps are needed for the OMA with n processors to execute A.

proof : Let R be the number of COMPS of an unrestricted version of A. Then $R \leq N$. Since each COMPS has at most two operands, r in Theorem 1 equals 2. By Theorem 1, OMA needs at most $3R \leq 3N$ steps to execute A.

Most of the problems that are of interest in a multiprocessing environment require the processors to know some information from other processors every few COMPS to continue processing. To be more specific, let us assume that for a particular algorithm A, the processors need to do c XFERS every o COMPS Let k be the average number of steps needed for each communication. Then the average communication overhead for A is

$$CO(A) = \frac{ck}{o + ck}$$

To illustrate this concept, let us pause to prove the following theorem.

Theorem 3 : For a mesh-connected machine of n^2 processing elements. The average number of steps for each communication is $N_{mesh} > n/2 - 1$.

proof : Let p_i be the probability that a random pair of PE's has distance i, i = 1,..,2n-2. We shall first prove that $p_i = p_{n-i}$, i = 1,...,n-1.
Define

$$T(x) = \begin{cases} x + n/2 & \text{if } x \le n/2; \\ x - n/2 & \text{if } x > n/2. \end{cases}$$

Notice that $T^2(x) = x$, so T is a one to one correspondence. Let $\{P_{jk}, P_{lm}\}$ be any pair of PE's with distance 1. Without loss of generality, we can assume j + k > l + m. Since T is a one to one correspondence, so is the following mapping:

$$R: \left\{P_{jk}, P_{lm}\right\} \longrightarrow \left\{P_{T(j),T(k)}, P_{lm}\right\}.$$

But since the distance of the pair $\{P_{T(i),T(j)}, P_{kl}\}$ has distance n - i, we conclude that the number of pairs with distance i equals the number of pairs with distance n - i. Therefore, the equation $p_i = p_{n-i}$ follows. Write A_k for the sum of the first k p_i's. i.e.

$$A_k = \sum_{i=1}^{k} p_i. \qquad k = 1,...,2n-2.$$

As a consequece of $p_i = p_{n-i}$, we get

$$A_k + A_{n-1-k} = A_{n-1}. \qquad k = 1,...,n-1. \qquad (*)$$

The average distance is given by

$$N_{mesh} = \sum_{i=1}^{2n-2} p_i i \qquad (K1)$$

By Abel's identity

$$\sum_{i=1}^{n} a_i b_i = S_n b_n - \sum_{i=1}^{n-1} S_i (b_{i+1} - b_i).$$

Where $S_k = \sum_{i=1}^{k} a_i$, k = 1,..,n. Eq. (K1) can be written as

$$N_{mesh} = 2n - 2 - \sum_{i=1}^{2n-3} A_i. \qquad (K2)$$

Now apply (*) to the above equation, we obtain

$$N_{mesh} > 2n - 2 - (n/2 + n - 1) = n/2 - 1.$$

Thus we completed the proof. ▪

Therefore, for a mesh-connected array,

$$CO_{mesh} = \frac{ck}{o + ck} > \frac{c(n/2 - 1)}{o + c(n/2 - 1)} = \frac{c(n-2)}{2o + c(n-2)}.$$

CO_{mesh} approaches 1 even for moderate n. Which means this type of machine is heavily loaded by the communication tasks. Such system cannot be made general-purpose even for any reasonable n.

For a logn stage network (lsn), k = logn. So

$$CO_{lsn}(A) = \frac{c\log n}{o + c\log n}$$

Again, CO_{lsn} tends to 1 as n becomes significantly large, which means the system spends most of its time executing communication steps for large n. This is better than the mesh network. But still, they are not efficient for a large number of processors on the average, even though they could be made special-purpose to utilize the best cases.

For the OMA, since k = 2,

$$CO_{OMA} = \frac{2c}{o + 2c}$$

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{pmatrix}$$

1, 5 33, 37	2, 6 34, 38	3, 7 35, 39	4, 8 36, 40
9, 13 41, 45	10, 14 42, 46	11, 15 43, 47	12, 16 44, 48
17, 21 49, 53	18, 22 50, 54	19, 23 51, 55	20, 24 52, 56
25, 29 57, 61	26, 30 58, 62	27, 31 59, 63	28, 32 60, 64

Figure 2: The matrix A wrapped-around in the OMA with four processors.

which is system size independent. **Hence an OMA machine could be made large without affecting the communications complexity.**

2.3 Storage schemes

For the OMA architecture, we consider two main storage schemes: wrap-around storage and block storage. Assume that we have an OMA with n processors, and let A = (a_{ij}) be an pn×qn matrix, which is to be stored in the OMA.

a) The wrap-around scheme

The elements a_{ij} are stored in memory module M_{lk}, Where

$$l = (i-1)n + 1$$
$$k = (j-1)n + 1$$

The content of memory module M_{ij} is shown below.

M_{ij}
a_{ij}
$a_{i,n+j}$
.
.
$a_{i,(q-1)n+j}$
$a_{n+i,j}$
.
.
$a_{n+i,(q-1)n+j}$
$a_{(p-1)n+i,(q-1)n+j}$

An example of how a 8×8 matrix is wrapped-around in a 4 processor OMA is shown in Figure 2.

b) The block scheme

We first write A in the block form. Each block is of size p×q. We have n×n such blocks. The block in the i-th row j-th column is stored in M_{ij}. Figure 3 shows the memory contents when storing a matrix in block mode.

III. MATRIX MULTIPLICATION

Matrix manipulations are frequently needed in solving linear systems of equations. Important matrix operations include matrix multiplication, L-U decomposition, matrix inversion and matrix transposition. L-U decomposition will be tackled in the next section. We shall not give an algorithm for matrix inversion explicitly. Instead, we shall give a method for solving linear systems via L-U

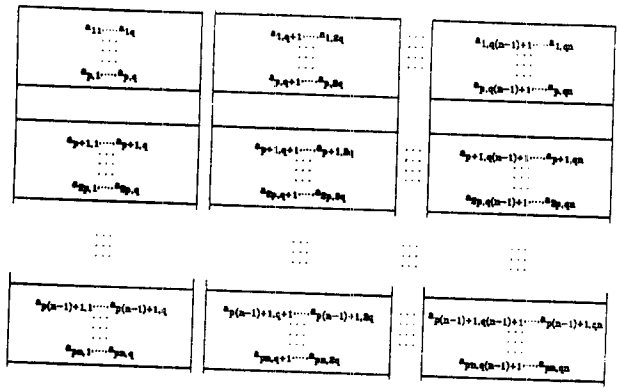
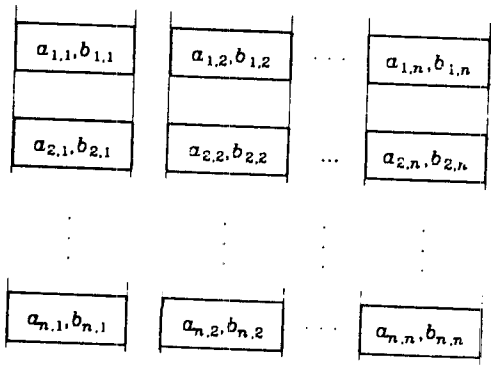


Figure 3: The block storage scheme.

decomposition and solve the resulting triangular systems. Matrix transpose is relatively easy for the OMA, and will not appear as an independent algorithm in this paper, but the reader should have no difficulty in deriving it out in either this or the next section.

Given two n×n matrices A and B, distributed with a_{ij}, b_{ij} in M_{ij}. As in the following configuration :



The product C = AB is given by the expression

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \qquad i,j = 1,..,n$$

Since processor P_k has access to both a_{ik} and b_{kj}, it can compute

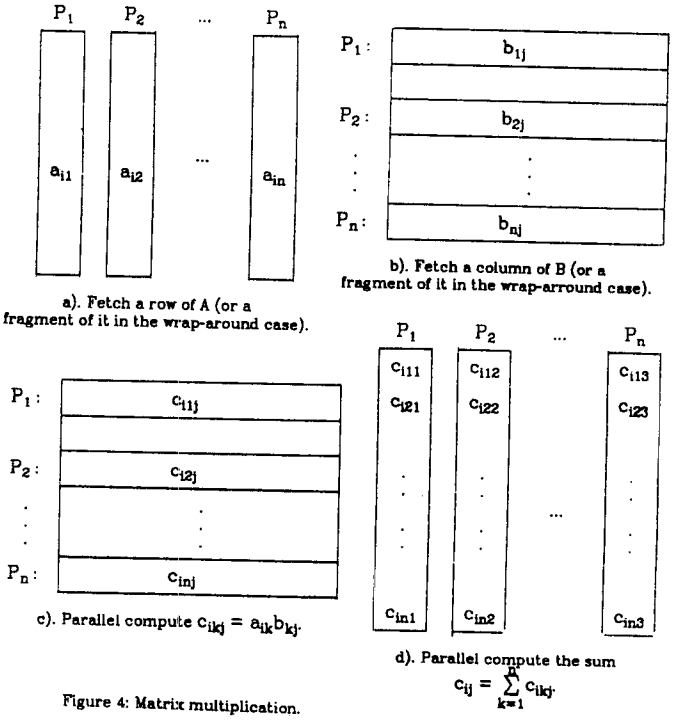
$$c_{ikj} = a_{ik}b_{kj}. \qquad i,j,k = 1,..,n$$

and put the result in M_{ik}. Next, observe that P_i has access to all M_{ik}, k = 1,..,n. The summation $c_{ij} = \sum_{k=1}^{n} c_{ikj}$ can then be computed by P_i and the result put in M_{ij}. A typical iteration is shown in Figure 4.

The result is easily expanded to the case when both A and B are of degree qn×qn. This time A and B are stored in a wrap-around manner. Since a_{ik} resides in M_{lm} and b_{kj} resides in M_{ms}, where i, l, k, m, j and s satisfy the following:

$$l = (i - 1) \bmod n + 1$$
$$m = (k - 1) \bmod n + 1$$
$$s = (j - 1) \bmod n + 1$$

P_m has access to both a_{ik} and b_{kj}. Therefore it can compute $c_{ikj} = a_{ik}b_{kj}$, and put the result in M_{lm}. Then since P_l has access to all c_{ikj} in M_{lm}, it can compute the sum $c_{ij} = \sum_{k=1}^{n} c_{ikj}$ and put it in M_{ls}.



a). Fetch a row of A (or a fragment of it in the wrap-around case).

b). Fetch a column of B (or a fragment of it in the wrap-arround case).

c). Parallel compute $c_{ikj} = a_{ik}b_{kj}$.

d). Parallel compute the sum $c_{ij} = \sum_{k=1}^{n} c_{ikj}$.

Figure 4: Matrix multiplication.

Algorithm 1 (MatMul) :

Step 1 : REPEAT Steps 2 to 5 for i = 1 to qn

Step 2 : CM fetch a fragment of a row of A in row memory $a_{i,j+1},...,a_{i,j+n}$.

Step 3 : RM REPEAT Steps 4, 5 for k = 1 to qn

Step 4 : RM fetch the corresponding fragment of column of B in a column memory $M_{j \bmod n,k \bmod n}$. $b_{j+1,k},...,b_{j+n,k}$.

Step 5 : RM compute $c_{i,j+1,k} = a_{i,j+1}b_{j+1,k}$, put it back in $M_{i \bmod n,k \bmod n}$.

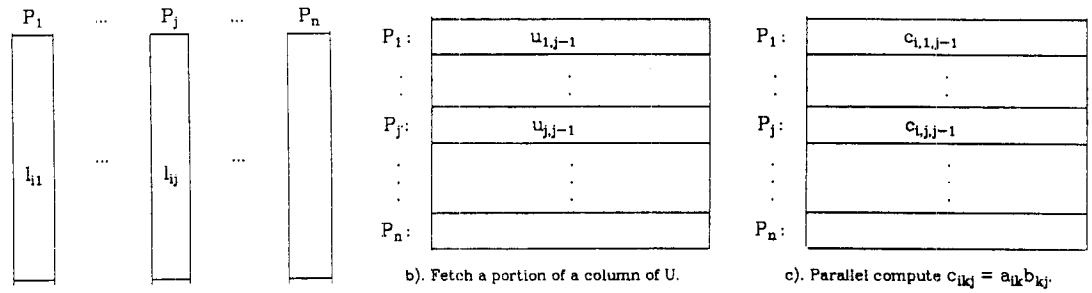
Step 6 : CM REPEAT Step 7 for k = 1 to qn

Step 7 : add all the c_{ijk}'s up to get c_{ij}. i.e. $c_{ij} = 0$; $c_{ij} = c_{ij} + c_{ijk}$.

Theorem 4 : The time complexity of matrix multiplication on an OMA is $O(q^3n^2)$, which has a linear speedup over the a uniprocessor conventional method.

proof : With P_k executing in parallel k = 1,..,n. to compute c_{ikj} i,j,k = 1,..qn. Each processor does two fetches and one multiplication. So q^3n^2 multiplication steps are required for the first phase of computation. To sum the c_{ikj}'s up, another q^3n^2 add steps are required, Thus $2q^3n^2$ is the total operation time.

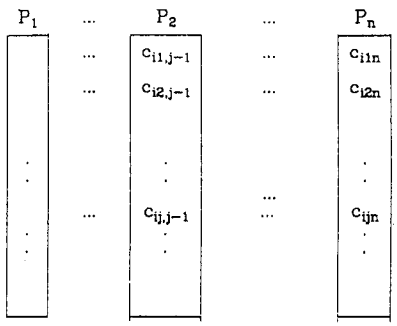
Let us analyze the communication problem for the matrix multiplication algorithm. From step 1 to step 5, we see that each fragment of a row is communicated for every qn multiplications. There are q^2n communications and q^3n^2 multiplications. For step 6 and step 7, there are no communications required, only q^3n^2 additions. So $o = q^3n^2 + q^3n^2 = 2q^3n^2$, $c = q^2n$, and



a). Fetch a portion of a row of L ($j < i$).

b). Fetch a portion of a column of U.

c). Parallel compute $c_{ikj} = a_{ik}b_{kj}$.



d). Parallel compute

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} c_{ikj}.$$

Figure 5: L-U decomposition.

$$CO_{OMA}(MatMul) = \frac{2c}{o + 2c} = \frac{2q^2n}{2q^3n^2 + 2q^2n} = \frac{1}{qn + 1}$$

Notice that qn is the size of the operand matrices. This means that for the algorithm given above, **the CO value is inversely proportional to the operand size.**

IV. L-U DECOMPOSITION

The classical Gaussian method for solving a linear system of equations

$$Ax = b$$

where A is an n×n matrix, x and b are n×1 vectors, is to transform the matrix to a lower triangular matrix via successive row transformations. Equivalently, one can decompose A into two trianglar matrices and then solve the resulting systems. The L-U decomposition problem is to find two matrices L and U of the form :

$$L = \begin{pmatrix} 1 & 0 & . & . & 0 \\ l_{21} & 1 & . & . & 0 \\ . & . & . & . & . \\ . & . & . & . & 0 \\ l_{n1} & l_{n2} & . & l_{n,n-1} & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_{11} & u_{12} & . & . & u_{1n} \\ 0 & u_{22} & . & . & u_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & . & 0 & u_{nn} \end{pmatrix}$$

such that A = LU. Then to solve Ax = b, we could, instead, solve two triangular systems

$$Ly = b$$

$$Ux = y.$$

In our OMA, A is initially spread out in the memory modules with a_{ij} in M_{ij}. The resulting L and U will be finally stored in the form :

$$\begin{pmatrix} u_{11} & u_{12} & . & . & . & u_{1n} \\ l_{21} & u_{22} & . & . & . & u_{2n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ l_{n1} & l_{n2} & . & . & l_{n,n-1} & u_{nn} \end{pmatrix}$$

The factorization takes n iterations. On the i-th iteration, the i-th row of U will be evaluated, and then the i-th column of L is evaluated, via the following formulae:

$$u_{1j} = a_{1j} \qquad j = 1,..,n$$

$$l_{j1} = a_{j1} / u_{11} \qquad j = 2,..,n$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \qquad j = i,..,n \qquad (U)$$

$$l_{ji} = \left[a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}\right] / u_{ii} \qquad j = i+1,..,n \qquad (L)$$

To compute (U), note that it is an inner product from elements in a row memory with elements in a column memory. As in matrix multiplication (Section III), it takes 2i computing steps. equation (L) is similar to (U) except that now we take advantage of the broadcasting feature. We first put u_{ii} in the i-th column memory by processor P_i, then proceed like (U). Finally, division by u_{ii} takes two additional steps. one for the broadcast, the other for the division. Figure 5 shows the diagram for a typical i-th iteration.

The result is easily generalized to the case of qn×qn matrices. The wrap-around storage scheme can be used and the argument is similar to the discussion in Section III. We can summarize the method into the following algorithm.

Algorithm 2 (LUDec) :

INPUT : a_{ij}, i,j = 1,..,qn. Wrap-around in OMA of n processors.

OUTPUT : lower triangular matrix L, upper triangular matrix U. Such that A=LU.

STEP 0 : Initialize L = 0, U = 0, c_{ijk} = 0, i=1,..,n, j,k=1,..,q.

STEP 1 : For i = 1 to qn do STEP 2 to STEP 14.

STEP 2 : For j = 1 to q do STEP 3 to STEP 5.

STEP 3 : CM mode. Parallel fetch

$$l_{i,(j-1)q+1}, \ldots, l_{i,(j-1)q+n}.$$

STEP 4 : RM mode. for k = 1 to q do STEP 5.
STEP 5 : Parallel multiply

$$c_{1kj} = l_{i,(j-1)q+1}u_{(j-1)q+1,i+(k-1)q}$$

.
.
.

$$c_{nkj} = l_{i,(j-1)q+n}u_{(j-1)q+n,i+(k-1)q}$$

STEP 6 : For k =1 to q do STEP 7
STEP 7 : Parallel

$$u_{i,(k-1)n+1} = a_{i,(k-1)n+1} - \sum_{l=1}^{n}\sum_{m=1}^{q} c_{l,(k-1)n+1,m}$$

.
.
.

$$u_{i,(k-1)n+n} = a_{i,kn} - \sum_{l=1}^{n}\sum_{m=1}^{q} c_{l,kn,m}$$

STEP 8 : CM mode. P_i pass u_{ii} to every memory module in i-th column.
STEP 9 : For j = 1 to q do STEP 10 to STEP 14.
STEP 10 : RM mode. Parallel fetch

$$u_{(j-1)q+1,i}$$
.
.
.
$$u_{(j-1)q+n,i}$$

STEP 11 : CM mode. for k = 1 to q do STEP 12.
STEP 12 : Parallel multiply

$$c_{1kj} = l_{i+(k-1)q,(j-1)q+1}u_{(j-1)q+1,i}$$

.
.
.

$$c_{nkj} = l_{i+(k-1)q,(j-1)q+n}u_{(j-1)q+n,i}$$

STEP 13 : For k =1 to q do STEP 14
STEP 14 : Parallel

$$l_{(k-1)n+1,i} = (a_{(k-1)n+1,i} - \sum_{l=1}^{n}\sum_{m=1}^{q} c_{l,(k-1)n+1,m})/\,u_{ii}$$

.
.
.

$$l_{(k-1)n+n,i} = (a_{kn,i} - \sum_{l=1}^{n}\sum_{m=1}^{q} c_{l,kn,m})/\,u_{ii}.$$

Theorem 5 : The LU decomposition of a qn×qn matrix takes $O(q^3n^2)$ steps.

proof : During the i-th iteration. Each of the u_{ij}, j = i,..,qn. requires 2i steps. Since the matrix is stored in a wrap-around manner. The qn - i + 1 evaluations have a parallelism of n. Thus

$$2i(qn - i + 1)/n \le 2qi.$$

steps are required to compute all u_{ij}, j = i,..,qn. It takes 2q more parallel steps to compute l_{ji}, j = i+1,..qn. So the total time for LUDec is :

$$T = \sum_{i=1}^{qn}(4qi+2q) = O(q^3n^2).$$

This yields an computation time of

$$o = \sum_{i=1}^{qn}(4qi + 2q) = 4q\frac{qn(qn + 1)}{2} + 2q^2n = 2q^3n^2 + 4q^2n.$$

with a total communication time given by :

$$c = 2q^2n$$

Thus the CO value becomes

$$CO_{OMA}(LUDec) = \frac{4q^2n}{2q^3n^2 + 4q^2n + 4q^2n} = \frac{2}{qn + 4}$$

V. TRIANGULAR SYSTEMS

Let A be an invertible lower n×n triangular matrix. The problem of solving a triangular system is to solve a system of linear equations of the form :

$$x_1 = c_{11}.$$
$$x_2 = c_{21}x_1+c_{22}.$$
.
.
$$x_n = c_{n1}x_1+...+c_{n,n-1}x_{n-1}+c_{nn}.$$

To achieve this in OMA, store the matrix $C = (c_{ij})$ with c_{ij} in M_{ij}. The proposed algorithm takes n iterations, and does the following in the i-th iteration :

1. Broadcast x_i in the i-th column memory.
2. Processor P_j for j > i computes $c_{ji}x_i+c_{jj}$ and the new value is assigned to c_{ij}.

Notice that at the end of the i-th iteration, $x_{i+1}=c_{j+1,j+1}$ is stored in $M_{j+1,j+1}$. The i-th iteration is illustrated in Figure 6.

The algorithm can be generalized without difficulty to the case of systems of size qn×qn.

Algorithm 3 (TRIS)

INPUT : $C = (c_{ij})$, i,j = 1,..,qn. with $c_{ij} = 0$ if i < j.
OUTPUT : Solution x_i i = 1,..,qn of

$$x_1 = c_{11}.$$
$$x_2 = c_{21}x_1+c_{22}.$$
.
.
$$x_n = c_{n1}x_1+...+c_{n,n-1}x_{n-1}+c_{nn}.$$

STEP 1 : For i = 1 to qn do STEP 2 to STEP 4.
STEP 2 : CM mode. P_i pass c_{ii} to the i-th column memory.
STEP 3 : For j = 1 to q do STEP 4.
STEP 4 : Parallel multiply and add

$$c_{(j-1)q+1,(j-1)q+1} = c_{(j-1)q+1,(j-1)q+1} + c_{(j-1)q+1,i}c_{ii}$$
.
.
.
$$c_{(j-1)q+n,(j-1)q+n} = c_{(j-1)q+n,(j-1)q+n} + c_{(j-1)q+n,i}c_{ii}$$

STEP 5 : For i = 1 to qn do $x_i = c_{ii}$.

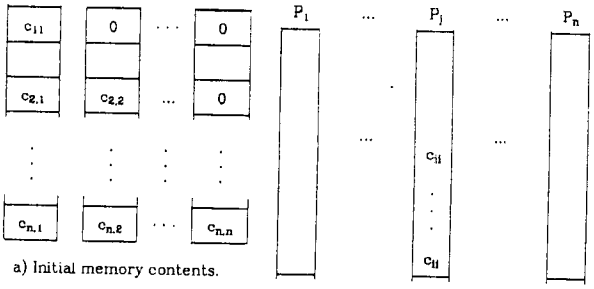
Thus we obtain the following theorem.

Theorem 6 : A triangular system of size qn×qn can be solved in the OMA using $O(q^2n)$ steps.

proof : In each iteration, there are at most qn computations of type ax + b, with a parallelism of n. So O(q) steps are needed in each iteration. There are qn iterations. Therefore the total time complexity is $O(q^2n)$.

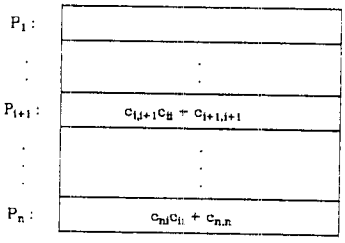
Corollary 1 : To solve a linear system of equations Ax=b on OMA, $O(q^3n^2)$ steps are sufficient.

proof : First apply L-U decomposition to A. By theorem 2, it takes $O(q^3n^2)$ steps. Then solve two triangular systems Ly = b, Ux = y. By theorem 3, each of them takes $O(q^2n)$ steps, giving a total of $O(q^3n^2)$ steps are required.



a) Initial memory contents.

b). Communicate c_{jj} to other procesors.



c). Parallel compute.

Figure 6: Triangular systems.

In view of algorithm 3, we did one communication for each multiplication and addition. So $\frac{o}{c} = \frac{2}{1} = 2$. The communication overhead for this algorithm is thus

$$CO_{OMA}(TRISY) = \frac{2c}{o + 2c} = \frac{2}{\frac{o}{c} + 2} = \frac{2}{2+2} = 1/2.$$

VI. PARTIAL DIFFERENTIAL EQUATIONS

Two methods will be discussed in this section. The iterative method and the transform method. The latter is also called the "direct method" in the literature.

The iterative method

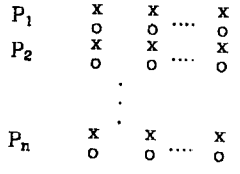
Consider the partial differential equation

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + B(x,y)\frac{\partial u}{\partial x} + C(x,y)\frac{\partial^2 u}{\partial y^2} + D(x,y)\frac{\partial u}{\partial y} + E(x,y)u = F(x,y).$$

which by finite difference approximation over a s×t mesh of points becomes

$$a_{p,q}u_{p,q-1} + b_{p,q}u_{p,q+1} + c_{p,q}u_{p-1,q} + d_{p,q}u_{p+1,q} + e_{p,q}u_{p,q} = f_{p,q}.$$

with the convention that elements of non-positive indexes are zero. Now the problem becomes one of solving a linear system of $2n^2$ variables. We shall assume that the serial algorithm to solve tri-diagonal systems (STRI) is available for use. For better system utilization, we choose a mesh of size 2n×n. Initially, every two rows were stored in one row of memory with each memory module containing two elements, as in the following configuration :



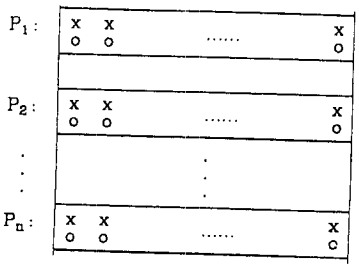
Now we update the x's and o's alternatively according to the following equations :

$$a_{p,q}u^{*}_{p-1,q} + b_{p,q}u^{*}_{p+1,q} + e_{p,q}u^{*}_{p,q} = f_{p,q} - c_{p,q}u_{p,q-1} - d_{p,q}u_{p,q+1} \quad (I1)$$

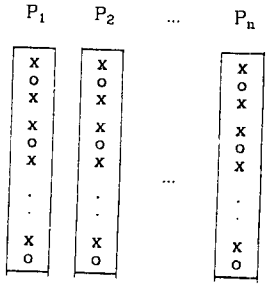
$$u^{new}_{p,q} = \omega u^{*}_{p,q} + (1-\omega)u^{old}_{p,q} = \omega(u^{*}_{p,q} - u^{old}_{p,q}) + u^{old}_{p,q} \quad (I2)$$

Where p = 1,3,..,2n-1, when updating odd rows; and p = 2,4,..,2n when updating even rows, q = 1,..,n, and $1 \le \omega \le 2$ is a chosen relaxation factor.

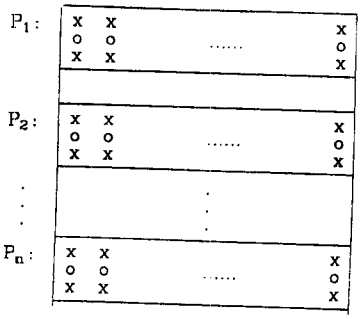
Notice that (I1) consists of n tri-diagonal systems (see Figure 7c). So if processor P_p can solve the p-th system by using STRI, then (I1) takes O(n) parallel steps. Before we can do this, we must make all the necessary data available to P_p. We must first move the data in the line below (or in the line above, depending on whether we are updating o's or x's) to the row memory of P_p (see Figure 7b). To do this we set the machine in CM mode, and pass the o's down one row (or x's up one row if updating o's). This takes n steps (if the shift option is available, then one step is sufficient). It then requires a total of O(n) steps.



a). Initial configuration.



b). When updating o's, move the x's adjacent to the upper row memory up.



c). Now each row can solve (6.1.3) use the serial tri-diagonal algorithm.

Figure 7: Iterative method of solving PDE

It takes three steps to update $u^{new}_{p,q}$ in view of (I2). So 3n parallel steps are enough to get through with (I2).

To check the termination condition, we let

$$\varepsilon = \omega(u^{*}_{p,q} - u^{old}_{p,q}) = u^{new}_{p,q} - u^{old}_{p,q} \quad (I3)$$

then $|\varepsilon|$, the absolute value of the first term of (I2) is the error. During the computaion of (I2), ε is obtained as a side product. Each processor replaces its own $|\varepsilon|$ by the bigger one as it computes $u^{new}_{p,q}$ along the row. Upon the end of each iteration, the $|\varepsilon|$ in P_i is the maximun among the errors in the i-th row. If any of the processors discovers its $|\varepsilon|$ is greater than the given tolerance, it issues a signal to the master processor to continue the process.

If none of the processors issues such a signal at the end of an iteration, the procedure terminates successfully.

This type of method is known as Successive Line Over Relaxation (SLOR). The approximate number of iterations needed to obtain the error within 10^{-p} is given by

$$t_{SLOR} \approx np/4.$$

The method introduced here applies also to a mesh points of size kn×ln. The block storage scheme can be adapted to reduce data movements. Under this scheme, only the last and the first rows in each row memory needs to be moved during the successive updates. No other parts of the method needs to be modified.

Algorithm 4 (IPDE)

INPUT : Initial approximation U = (u_{ij}), and coefficient matrices

A = (a_{ij}), B = (b_{ij}), C = (c_{ij}), D = (d_{ij}), E = (e_{ij}), F = (f_{ij}).

i = 1,..,kn, k is even; j = 1,..,ln.
And tolerance ε_0.

OUTPUT : Final approximation u_{ij}.

STEP 1 : flag = STOP;

STEP 2 : CM mode. For i = 1 to n-1 do STEP 3.

STEP 3 : Move the ki-th row of U to the (i+1) -th row memory.

STEP 4 : RM mode. Update all the odd rows via eqs. (I1) and (I2). and e according to (I3).

STEP 5 : CM mode. For i = 1 to n-1 do STEP 6.

STEP 6 : Move the (ki+1)-th row of U to the i-th row memory.

STEP 7 : RM mode. Update all even rows via eqs. (I1) and (I2). and ε according to (I3). If $\varepsilon > \varepsilon_0$ for some processor P_p, then P_p set the shared variable flag = CONTINUE.

STEP 8 : If flag = CONTINUE then goto STEP 1 else exit.

Theorem 7 : The algorithm presented above has a linear speedup over its corresponding serial uniprocessor version.

proof : In each iteration we move n-1 rows (either up or down according to even or odd row update). This takes l(n-1) parallel steps. Using Crout's method [2] to solve (I1), it will take 8ln-3 steps. Each processor has to solve k/2 of them. Thus, the time complexity becomes (8ln-3)k/2. 5ln more steps are needed to compute $u_{p,q}^{new}$ and check the termination condition. Therefore the total time is

$$T = l(n-1) + (8ln-3)k/2 + 5ln = O(lkn).$$

The serial time to do this would be O(kn×ln). Therefore, the theorem follows.

To compute the communication overhead, we see that each iteration consists of l(n-1) communications and 4lkn + 5ln - 3k/2 operation steps, So

$$CO_{OMA}(TPDE) = \frac{2l(n-1)}{4lkn + 5ln - 3k/2 + 2l(n-1)} \approx \frac{1}{2k+3}.$$

Note that CO_2 depends only on k, which points towards a better efficiency in grids with more rows than columns.

The Transform Method

Consider the difference equation (I1). We take the special case when

$$a_{p,q} = b_{p,q} = c_{p,q} = 1, \quad e_{p,q} = -1.$$

So (I1) reduces to

$$u_{p,q-1} + u_{p,q+1} + u_{p-1,q} + u_{p+1,q} = f_{p,q} \qquad p,q=1,..,n \quad (T1)$$

Let $v^{s,t}$ be the double Fourier transform of $u_{p,q}$, and $g^{s,t}$ be that of $f_{p,q}$.

$$v^{s,t} = \frac{1}{n^2}\sum_{p,q=0}^{n-1} u_{p,q} e^{-2\pi i(sp+tq)/n}$$

$$g^{s,t} = \frac{1}{n^2}\sum_{p,q=0}^{n-1} f_{p,q} e^{-2\pi i(sp+tq)/n}$$

By taking the double Fourier transform on both sides of (T1), we get

$$\left[2\cos(2\pi s/n)+2\cos(2\pi t/n)-4\right]v^{s,t} = g^{s,t}$$

To solve (T1). we first take the double Fourier transform of $f_{p,q}$, which according to [1], takes O(logn) steps. Then each entry is divided by the expression in the square bracket. Finally, a double inverse Fourier transform is performed. The first step takes O(logn). Each division takes a constant time. There are n divisions for each processor, so O(n) is needed for the division step. The last step takes the same amount of time as the first step, (i.e. O(logn)). So the total time is O(n). Since the Fourier transform algorithm in [1] works for kn×kn mesh points, this direct method applies to the kn×kn case as well. So actually we proved Theorem 8.

Theorem 8 : The equation given in (T1) can be solved in time $O(k^2n)$.

The method described above may be summarized into the following simple algorithm.

Algorithm 5 (TPDE)

INPUT : F = (f_{ij}), i,j = 1,..,kn;

OUTPUT : The solution u_{ij} of (T1).

STEP 1 : $g^{s,t}$ = Double_F-transform(f_{pq}).

STEP 2 : $v^{s,t} = g^{s,t}/[2\cos(2\pi s/n) + 2\cos(2\pi t/n) - 4)]$.

STEP 3 : u_{ij} = Double_Invers_F_transform($v^{s,t}$).

VII. POLYNOMIAL EVALUATION

Let $P(x) = \sum_{k=0}^{n^2-1} a_k x^k$ be a polynomial of degree $n^2 - 1$. and x_0 be any given number. To compute $P(x_0)$, a modified Horner's method is appropriate for the OMA.

Rewrite P(x) in the form :

$$P(x) = \sum_{i,j=1}^{n} a_{ij} x^{(i-1)n+j-1}$$
$$= \sum_{j=1}^{n}\left(\sum_{i=1}^{n} a_{ij} x^{(i-1)n}\right)x^{j-1}$$

Hence the problem of evaluating a polynomial of degree $n^2 - 1$ is reduced to the evaluation of the following n + 1 polynomials of degree n.

$$b_j = \sum_{i=1}^{n} a_{ij}(x_0^n)^{i-1} \qquad j=1,..,n \qquad (H1)$$

and

$$P(x_0) = \sum_{j=1}^{n} b_j x_0^{j-1}. \qquad (H2)$$

Observe that the n equations in (H1) can be executed in parallel. Since for fixed j, a_{ij} are in the same column of memories, we let processor P_j compute the j-th equation in (H1). By using Horner's method to evaluate (H1) at x_0^n, 4n parallel steps plus logn steps to compute x_0^n are

needed. To evaluate (H2), another 2n steps are required. So a total of 4n + logn steps solves the problem, whereas $2(n^2 - 1)$ steps are neccesary if done serially. Thus a linear speedup is achieved. Figure 8 shows a schematic diagram of the algorithm.

The method can be generalized to the case when P(x) is of degree $(qn)^2 - 1$. This time a_{ij} i,j = 1,..,qn are stored in a wrap-around fashion. Equation (H1) now becomes

$$b_j = \sum_{i=1}^{qn} a_{ij}(x_0^p)^{i-1} \qquad j = 1,..,qn \qquad \text{(H2')}$$

and (H2) becomes

$$P(x_0) = \sum_{j=1}^{qn} b_j x_0^{j-1}. \qquad \text{(H2')}$$

(H1') differs from (H1) in that here we have qn evaluations instead of n, and each polynomial is of degree qn. Since wrap-around storage is employed, the coefficients of each polynomial in (H1') are still in the same column memory. Therefore the time complexity to evaluate (H1'), with every processor executing in parallel becomes :

$$q(2qn + logn) + 2qn = 2q^2n + logn + 2qn = O(q^2n)$$

Since the serial time is $2(qn)^2$, we arrive at the following theorem.

Theorem 9 : A polynomial of degree $(qn)^2 - 1$ can be evaluated in $O(q^2n)$ steps.

Algorithm 6

INPUT : A = (a_{ij}), i,j = 1,..,qn. the coefficient matrix of P(x). wrap-arounded in the OMA of n processors.

OUTPUT : a = $P(x_0)$.

STEP 1 : Parallel. P_i evaluates $b_{i,qn} = a_{i,qn}$, $b_{ij} = a_{ij} + b_{i,j+1}x_0$.

STEP 2 : P_1 evaluates $c_{qn} = b_{qn,1}$, $c_i = b_{i1} + c_{i+1}x_0^{qn}$.

STEP 3 : a = c_1.

As two consequences of the polynomial evaluation, we shall present the iterative methods of finding zeroes of a given polynomial.

1. Newton's method

Let x_0 be any given initial approximation and ε be the given tolerance. For i≥1 keep evaluating the following expression

$$x_i = x_{i-1} - \frac{P(x_{i-1})}{P'(x_{i-1})}$$

until $|x_i - x_{i-1}| < \varepsilon$.

We have an algorithm to evaluate P at any point. To evaluate P' note that

$$P'(x) = \frac{\left(\sum_{i,j=1}^{qn} a'_{ij}x^{(i-1)n+j-1}\right)}{x.} \qquad \text{(N)}$$

Where $a'_{ij} = [(i-1)n + j-1]a_{ij}$.

Equation (N) doesn't make sense when x = 0. To take care of this problem, we first test x', the point where we are evaluating. If it is 0, then $P'(x') = 2a_{12}$. Otherwise, we go ahead and compute a'_{ij}, use these as coefficients, and evaluate at x'. Finally, we divide by x'.

To check the termination condition, we can use the same strategy as we did in the iterative method of solving PDEs.

2. The bisection method

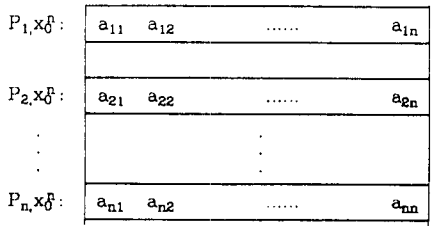
Let a and b be any two numbers with a < b, and let ε be a given tolerance. Suppose P(a)P(b) < 0. Then by the intermediate value theorem of continuous functions, there is an x_0 in (a,b) such that $P(x_0) = 0$.

Now let $c = \frac{b-a}{2}$. One of the following must be true

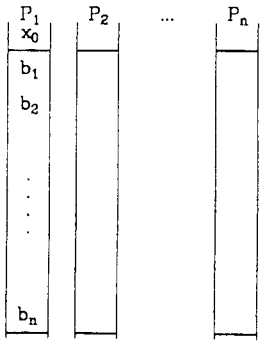
1) P(a)P(c) < 0.
2) P(b)P(c) < 0.
3) P(c) = 0.

If 1) is true, and |c - a| > ε, then we let b <-- c. Repeat the process. If 2) is true, and |b - c| > ε, then we let a <-- c. Repeat the process. Finally, if 3) is true, we found the root!.

This method is nothing more than evaluating polynomials and making assignments.



a). Using Horner's method for each row evaluate at x_0^p.



b). Using Horner's method for column evaluate at x_0.

Figure 8: Polynomial evaluation.

VIII CONCLUSIONS

The analysis of computer architectures and computer algorithms has always been concentrated on the speed, cost and the ease of application and implementation. For the issue of parallel algorithms, speed is obtained by utilizing all of the processors while reducing unnecessary communications. In spite of the fact that the communication pattern and complexity varies from algorithm to algorithm, the diameter could serve as a rough measure, and is ideally 1. But a unit diameter implies that the network is completely connected, which is cost-prohibitive. Therefore 2 is the practical minimum. In this sense, the OMA machine proposed here is optimal.

We have proven the correctness of the algorithms presented in this paper. Compared to algorithms for most proposed parallel the ones studied here are simple, both for understanding and implementation, and exhibit linear speedups in the OMA machine. The question of how the OMA behaves compared to other parallel machines was answered in the begining of this paper: it can't be worse than any other system with the same amount of processors beyond a factor of 3. The proof of this provides a way of producing algorithms for OMA out of algorithms for other machines.

References

[1] P.S Tseng, K. Hwang and P. K. Kumar *A VLSI-based multiprocessor architecture for implementing Parallel algorithms*. Proceedings of the 13-th International Conference on Parallel Processing, Aug. 1985.

[2] Richard L. Burden, J. Douglas Faires and Albert C. Reynolds. (1978), *Numerical Analisis*(second edition). PWS publishers, Mass.

[3] Gulbin Ezer, Brad Hoyt, Sandeep Sen, J.S. Sreekanth and Isaac Scherson *A parallel processing architecture for image generation and processing*. Preliminary report, ECE 84-20, University of California, Santa Barbara. Aug. 1984.

[4] R. W. Hockney and C. R. Jesshope. (1981), *Parallel Computers*. Adams Hilger Ltd, Bristol.

[5] E. Horowitz and S. Sahni. (1978), *Fundamentals of Computer Algorithms*. Computer Science Press.

[6] K. Hwang and F. Briggs. (1984), *Computer Architecture and Parallel Processing*. McGraw Hill.

[7] Louis A. Hageman and David M. Young. (1981) *Applied Iterative Methods*. Academic Press.

[8] Roland C. Le Bail. (1972), *Use of fast Fourier transforms for solving partial differential equations in physics*. Journal of Computational Physics 9, 440-465.

[9] Paul N. Swarztrauber. (1973), *The direct solution of the discrete Poisson equation on the surface of a sphere*. Journal of Computational Physics 15, 46-54.

[10] Roland A. Sweet. (1973), *Direct Method for the solution of Poisson equation on a staggered grid*. Journal of Computational Physics 12, 422-428.

[11] Vargar R. S. (1962), *Matrix iterative Analysis*. Prentice Hall, England Cliffs. N. J.

[12] Alfred N. Aho, Hopcroft and Ullman. (1982), *Data Structures and Algorithms*. Addison - Wesley.

[13] Alfred N. Aho, Hopcroft and Ullman. (1974), *The design and Analysis of Computer Algorithms*. Addison - Wesley.

[14] Leon S. Lasdon. (1970), *Optimisation Theory for Large Systems*. MaCmillan.

[15] Knuth. (1973) *The Art of Computer Programing*. volume 1 (second edition). Addison - Wesley.

[16] Knuth. (1981) *The Art of Computer Programing*. volume 2 (second edition). Addison - Wesley.

[17] Jean Loup Baer. (1980), *Computer Systems Architecture*. Computer Science Press.

[18] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh and D.L. Slotnik. *The Illiac IV system*. Proc. IEEE. vol. 60, no. 4, Apr. 1972, pp. 369-388.

[19] K.E. Batcher. *The flip network in STARAN*. Int'l. Conf. Parallel proc. Aug. 1976. pp.65-71.

[20] K.E. Batcher. *Design of a Massively Parallel Processor*. IEEE Trans. on comp., C-29, Sept. 1980. pp. 836-840.