Parallel Computing 4 (1987) 17–31
North-Holland

Matrix algorithms on a hypercube I: Matrix multiplication *

G.C. FOX and S.W. OTTO

Physics Department, California Institute of Technology, Pasadena, CA 91125, U.S.A.

A.J.G. HEY

Physics Department, The University, Southhampton SO9 5NH, United Kingdom

Received December 1985

Abstract. We discuss algorithms for matrix multiplication on a concurrent processor containing a two-dimensional mesh or richer topology. We present detailed performance measurements on hypercubes with 4, 16, and 64 nodes, and analyze them in terms of communication overhead and load balancing. We show that the decomposition into square subblocks is optimal. C code implementing the algorithms is available.

Keywords. Matrix multiplication, Caltech Hypercube, performance analysis, communication overhead, load balancing.

1. Introduction

This paper is designed to the be the first of a set on matrix algorithms for the hypercube. Clearly matrix operations are the heart of many important scientific computations and we need to build up a library of a concurrent matrix algorithms that are efficient and easy to use. We also need to build algorithms that run on a variety of concurrent processors and scale well to future machines with very many (up to 10^4–10^5) nodes. Our explicit implementations will use the Caltech Hypercube [1,5,6,15] but our analysis applies to any machine of this type (MIMD, distributed memory) that contains at least a two-dimensional mesh topology.

We intend a set of papers covering full matrix inversion (LU decomposition), banded matrix decomposition and the determination of eigenvalues for full matrices. In this paper we discuss a much simpler problem: that of matrix multiplication. We cover it in detail because it illustrates many of the essential points of the more complicated matrix operations and multiplication provides a simple environment in which to study them.

Our theoretical analysis is backed up by a complete experimental study of the algorithm which we have implemented in the C programming language on the Caltech hypercube. This code may be obtained by interested readers from: Caltech Concurrent Computation Program, Mailcode 159-79, Pasadena, CA 91125, U.S.A. A nominal charge is made to cover expenses.

Matrix algorithms for parallel or concurrent machines have been studied for many years—the work on systolic arrays [10] being some of the most relevant for our purposes. Other important work has come from the groups at Yale [9], JPL [17], Oak Ridge [7,8] and INTEL [11].

* The research reported here was supported in part by Department of Energy grants DE-AS03-ER13118, DE-FG-03-85ER25009, and by the Parsons Foundation and Systems Development Foundation. S. Otto holds a Bantrell Research Fellowship at Caltech.

The algorithms discussed here were first presented in detail in an unpublished memo by Fox [2]. We emphasize that we feel the importance of our research lies not in a novel algorithm but rather in a detailed theoretical performance analysis [3] which is backed up by reliable measurements on optimized code for the hypercube.

In Section 2, we describe the general decomposition while in Section 3, we detail this for the optimal choice of Square Subblocks; this section also includes a theoretical performance analysis. Section 4 mirrors Section 3 for a rectangular decomposition including as a special case that where the processors hold complete rows or columns.

In Section 5, we confront the algorithms with reality and analyze explicit implementations on the Caltech Mark II hypercube. Section 6 contains our conclusions. Appendix A describes an important technical issue in the communication while appendix B summarizes the essential hardware features and performance of the Caltech hypercube.

2. Decomposition

We are considering the class of concurrent computers commonly termed MIMD (Multiple Instruction, Multiple Data) with distributed (not shared) memory. For the algorithm discussed here, we can consider each node as running a single process with processes communicating via messages. Decomposition involves breaking up the underlying data—in this case initial and final matrices—into parts and associating one part of each matrix with each processor. We will find that this decomposition is not static and evolves as the algorithm progresses.

For the main body of the text we will only need that the processors have at least a two-dimensional periodic mesh interconnect. The hypercube includes this topology into which it is mapped by the usual binary Grey code scheme [14]. The mesh topology is sufficient for an efficient implementation of matrix multiplication. Appendix A discusses a partial extension to the use of the full hypercube topology.

We should also note that we only consider the even hypercubes which can be mapped into a square mesh; the extension of our work to the odd cubes (e.g., a 32 node 5-cube mapped into a 4×8 array is straightforward using the techniques described in Section 4.

We are interested in performing the multiplication,

$$C = A \cdot B \tag{1}$$

where C, A, and B are full, $M \times M$ matrices (for simplicity, we will only discuss the multiplication of square matrices).

We assume that the matrices A and B are decomposed as subblocks, shown in Fig. 1. We will demand that the product, C, will, after the matrix multiplication, end up decomposed in

$\hat{A}_{00}$	$\hat{A}_{01}$	$\hat{A}_{02}$	$\hat{A}_{03}$
$\hat{A}_{10}$	$\hat{A}_{11}$	$\hat{A}_{12}$	$\hat{A}_{13}$
$\hat{A}_{20}$	$\hat{A}_{21}$	$\hat{A}_{22}$	$\hat{A}_{23}$
$\hat{A}_{30}$	$\hat{A}_{31}$	$\hat{A}_{32}$	$\hat{A}_{33}$

Fig. 1. The square subblock decomposition of a matrix, A, onto a 4×4 mesh of processors. The hats refer to submatrices: $\hat{A}_{lk} = A_{ij}$, with $\frac{1}{4}Ml \leqslant i < \frac{1}{4}M(l+1)$, $\frac{1}{4}Mk \leqslant j < \frac{1}{4}M(k+1)$.

the same way so that it could perhaps form the input for further processing steps. The algorithm will first be described in terms of square subblocks. The extension of the algorithm and its performance analysis to rectangular subblocks and their extreme limit, a pure row or column decomposition (1D ring), will come next.

Note that we are assuming the input matrices A and B already reside in the Hypercube decomposed in the two-dimensional fashion—our timings do not take into account any loading of A or B. This is natural if A and B came from previous processing steps of some larger algorithm within the hypercube. On the other hand, if one is interested in using the hypercube as a 'flow through' matrix multiplier, some additional hardware enhancement is required to reach acceptable I/O performance levels. This is a general problem of essentially all parallel computers and can clearly be solved by a parallel I/O disk system. It offers no special demands in this regard and we will not discuss it here.

3. The square subblock decomposition

3.1. The algorithm

The algorithm is illustrated in Fig. 2 for the case of a 4×4 mesh of processors. The square submatrices are denoted by hats: $\hat{C}_{00}$ denotes the matrix C_{ij} with $0 \leqslant i < \frac{1}{4}M$, $0 \leqslant j \leqslant \frac{1}{4}M$, $\hat{C}_{lk}$ has elements C_{ij}, with $\frac{1}{4}Ml \leqslant i < \frac{1}{4}M(l+1)$. $\frac{1}{4}Mk \leqslant j < \frac{1}{4}M(k+1)$. Square submatrices can be manipulated as if they were single elements of the matrix. That is,

$$\hat{C}_{lk} = \sum_n \hat{A}_{ln} \cdot \hat{B}_{nk}$$

where the multiplication in the sum is actually matrix multiplication of the square submatrices, $\hat{A}_{ln}$ and $\hat{B}_{nk}$.

The algorithm proceeds as follows:

Step 1 (see Fig. 2(a)). Broadcast (in a pipelined fashion) the diagonal subblocks of A in a horizontal direction, so that all processors in the first 'row' have a copy of $\hat{A}_{00}$, all processors in the second 'row' have a copy of $\hat{A}_{11}$, and so on.

Step 2 (see Fig. 2(b)). Multiply the copied A subblocks into the B subblocks currently residing in each processor.

Step 3 (see Fig. 2(c)). 'Roll' the B subblocks vertically.

Step 4 (see Fig. 2(d)). Do a horizontal broadcast (as in step 1) of the 'diagonal +1' subblocks of A (see the figure).

Step 5 (see Fig. 2(e)). Multiply the copied A subblocks into the currently residing B subblocks.

Continue this pattern until B has 'rolled' completely around the machine. From the figure, it is clear that the product, C, ends up being constructed and decomposed in the correct fashion.

3.2. Performance analysis for square subblocks

Suppose we multiply $M \times M$ matrices on an $\sqrt{N} \times \sqrt{N}$ array of nodes. The size of the sublocks in each processor is then $m \times m$, where $m = M/\sqrt{N}$. The 'broadcast–multiply–roll' cycle of the algorithm is repeated $\sqrt{N}$ times. For each such cycle:

The time taken to broadcast the A submatrix is

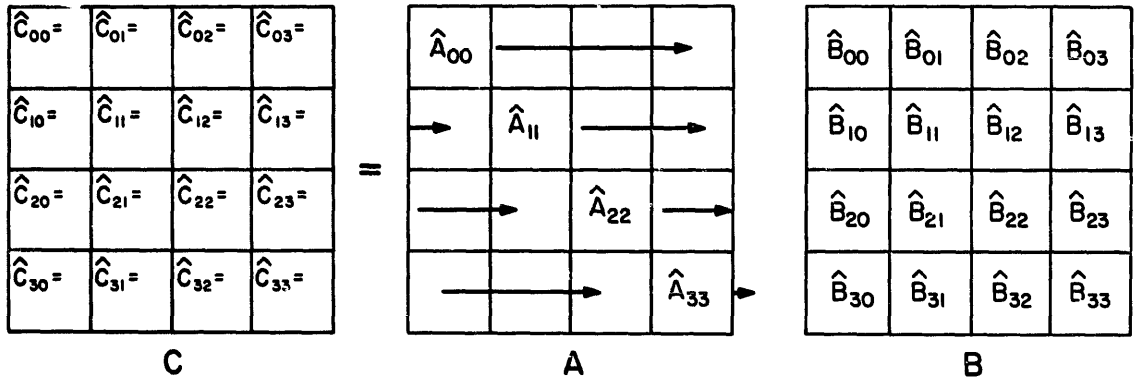
$$m^2 t_{\text{comm}} + (\sqrt{N} - 1) t_{\text{start}}. \tag{2}$$

We are assuming this is done as a linear pipeline (see Appendix A for a discussion of this) and t_{comm} = time to pass a floating point word between processors, t_{start} = startup time of the pipeline per step of the pipe.

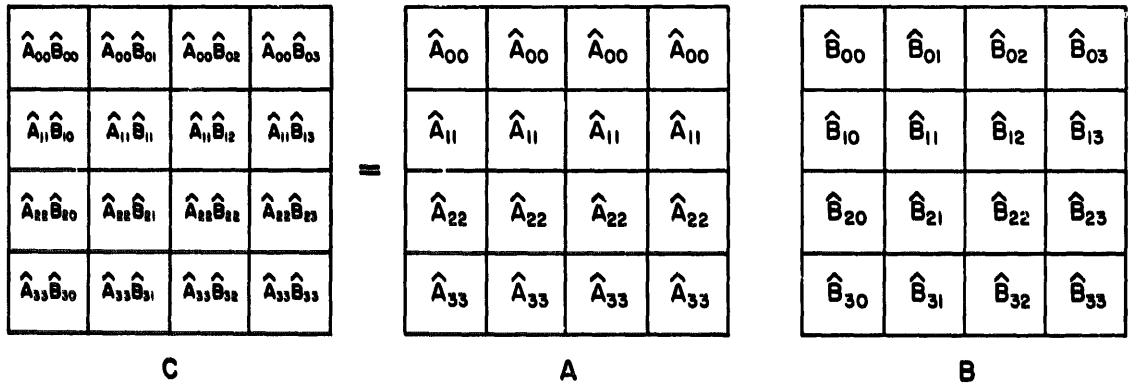
The time taken to 'roll' B is

$$m^2 t_{comm}. \tag{3}$$

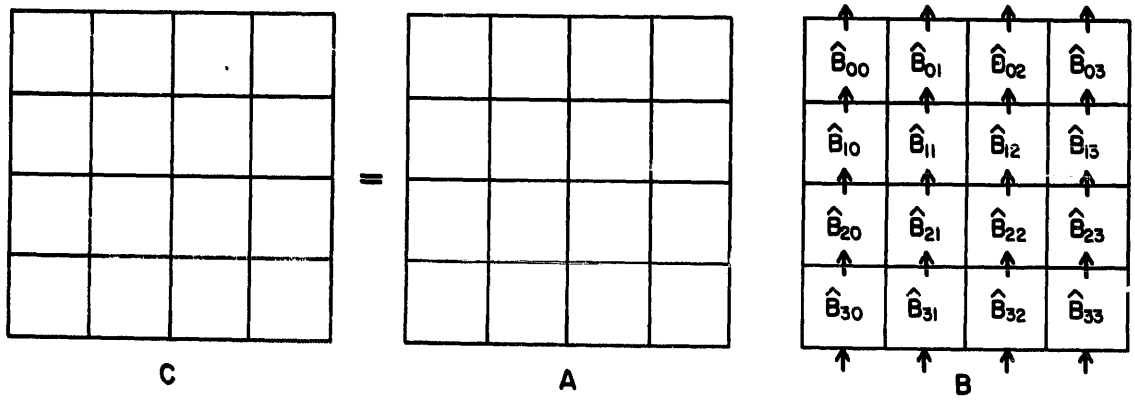
Note that, in (2) and (3), we are assuming that there is little startup time in initiating a message



(a)



(b)



(c)

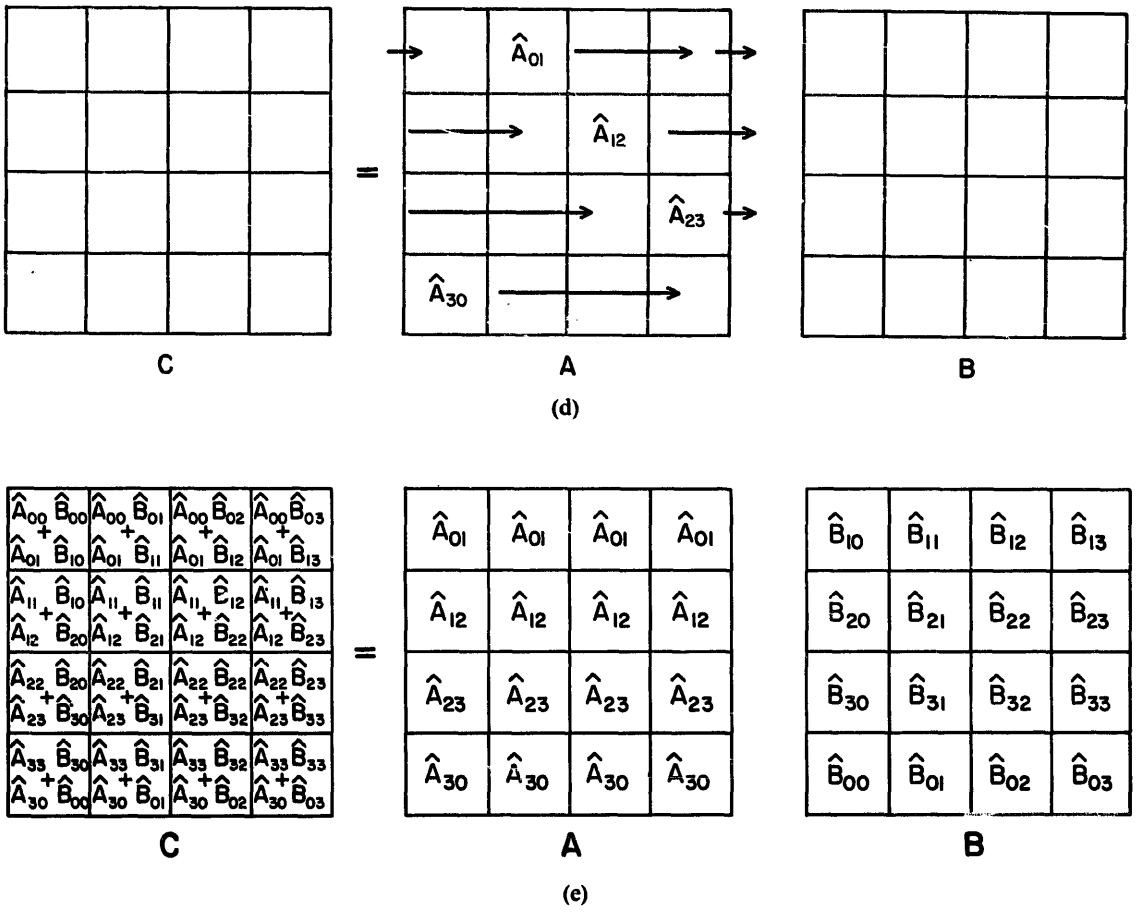


Fig. 2. The square subblock algorithm for a 4×4 machine. See the text for an explanation of each step.

between two processors. This is an accurate assumption for the Caltech/JPL machines—see Appendix B [4]. [1]

Finally, the time actually to compute the submatrix product (to be added to C) is

$$2m^3 t_{\text{flop}} \tag{4}$$

where t_{flop} is the time to do a floating point multiply or add (these two times are very nearly equal in our hardware—see Appendix B). Therefore, the total computation time of the matrix multiplication should be

$$T = \sqrt{N}\left(2m^3 t_{\text{flop}} + 2m^2 t_{\text{comm}} + (\sqrt{N} - 1)t_{\text{start}}\right), \tag{5}$$

$$T = \frac{2M^3 t_{\text{flop}}}{N} + \frac{2M^2 t_{\text{comm}}}{\sqrt{N}} + \sqrt{N}(\sqrt{N} - 1)t_{\text{start}}.$$

The first term is the expected 'perfect linear speedup' term while the last two terms represent communication overheads. Before presenting actual timing results for this algorithm, let us first discuss the case of rectangular subblocks, or column (1D ring) decomposition.

[1] The inclusion of a startup term in (2) and (3) (appropriate for some machine architectures) would modify details of our formulas, but would not modify our main result: two-dimensional decompositions would still be more efficient than one-dimensional decompositions.

4. Rectangular subblock decomposition

4.1. The algorithm

We now wish to extend the previous algorithm to the case where the submatrices in each processor are no longer square. The situation is as shown in Fig. 3.

An $M \times M$ matrix is decomposed onto an $N_0 \times N_1$ processor array where, for simplicity of presentation, we restrict ourselves to the case $N_0 \leq N_1$. The submatrix in each processor is $m_0 \times m_1$ where $m_0 = M/N_0$, $m_1 = M/N_1$. Furthermore, assume that the 'aspect ratio', $r = m_0/m_1$, is an integer, so that an integral number of square subblocks of size $m_1 \times m_1$ fits in each processor. The issue of 'padding' a matrix which may not perfectly fit the parallel machine will be discussed later.

The rectangular algorithm proceeds by the same strategy as before—the only difference is now that each processing node 'time shares' itself among several square subblocks.

4.2. Performance analysis for rectrangular subblocks

The timing analysis of this algorithm is as follows:

The broadcast–multiply–roll cycle of the algorithm is repeated N_1 times. For each such cycle, the time taken to do the horizontal broadcast is

$$m_1^2 t_{\text{comm}} + (N_0 - 1)t_{\text{start}}. \tag{6}$$

The time to multiply subblocks is

$$r \cdot 2m_1^3 t_{\text{flop}} = 2m_1^2 m_0 t_{\text{flop}} \tag{7}$$

and the time to roll vertically is

$$m_1 m_0 t_{\text{comm}}. \tag{8}$$

Therefore, the total computation time for the rectangular subblock algorithm is

$$T = N_1\left[2m_1^2 m_0 t_{\text{flop}} + t_{\text{comm}}(m_1^2 + m_1 m_0) + (N_0 - 1)t_{\text{start}}\right], \tag{9}$$

$$T = \frac{2M^3 t_{\text{flop}}}{N} + N_1\left[t_{\text{comm}}(m_1^2 + m_1 m_0) + (N_0 - 1)t_{\text{start}}\right].$$

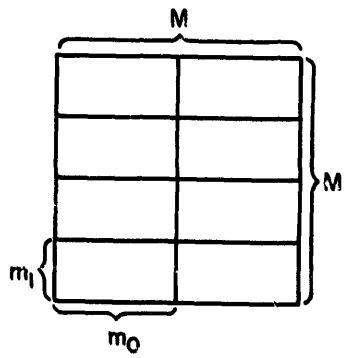


Fig. 3. An $M \times M$ matrix decomposed onto a 2×4 processor mesh ($N_0 = 2$, $N_1 = 4$). $m_0 = M/N_0 = \frac{1}{2}M$, $m_1 = M/N_1 = \frac{1}{4}M$.

We would now like to answer the question: what is the optimal choice of the aspect ratio, r, holding the total number of processors $N_0N_1 = N$, and the matrix size, M, fixed? The communication overhead can be read off (9) as

$$N_1 t_{\text{comm}}(m_1^2 + m_1 m_0) + N_1(N_0 - 1)t_{\text{start}}. \tag{10}$$

If the 1 is ignored relative to N_0, it is seen that the second term, the pipeline startup time, is a fixed cost. This is further justified by the fact that, at least for the Caltech/JPL hypercubes running the Crystalline Operating System, t_{start} is comparable to t_{comm}, so that the pipeline startup time is typically a very small effect. Continuing, we wish to minimize:

$$t_{\text{comm}} N_1(m_1^2 + m_1 m_0) = t_{\text{comm}} M(m_1 + m_0). \tag{11}$$

Since M is held fixed, the minimum is found at $m_1 = m_0$, or $r = 1$.

Matrix multiplication with square subblocks is most efficient. This conclusion does not yet extend to the case of a pure row storage algorithm (one-dimensional ring composition), since, in that case, the timing formula (9) does not hold. For pure row or column storage schemes, there is no broadcast step in the algorithm, so (9) is modified to

$$T_{\text{row,column}} = \frac{2M^3}{N} t_{\text{flop}} + N t_{\text{comm}} \frac{M^2}{N}. \tag{12}$$

4.3. Comparison of square subblocks versus row / column

To finish the performance analysis, we will write down the efficiencies for the square subblock and the pure row algorithms. Define efficiency as the speedup per processing node:

$$\epsilon = \frac{1}{N} S = \frac{1}{N} \frac{\text{time on 1 processor}}{\text{time on } N \text{ processors}}. \tag{13}$$

For the subblock algorithm, the efficiency can be read off of (5):

$$\epsilon_{\text{block}} = \frac{1}{1 + \frac{1}{\sqrt{N}} \frac{t_{\text{comm}}}{t_{\text{flop}}} + \delta} \tag{14}$$

where we have written 'n' for the number of matrix elements per node ($n = M^2/N$) and where δ is a relatively small term representing the pipeline startup. In the same notation, the efficiency for the pure row scheme is read off from (12) as

$$\epsilon_{\text{row}} = \frac{1}{1 + \frac{1}{\sqrt{n}} \frac{t_{\text{comm}}}{t_{\text{flop}}} \frac{1}{2}\sqrt{N}}. \tag{15}$$

The overhead for the pure row algorithm is larger by a factor $\frac{1}{2}\sqrt{N}$, where N is the number of processing nodes.

5. Timings on the Caltech hypercube

5.1. Load balanced decompositions

To verify that our proceeding theoretical analysis of matrix multiplication algorithms was sound, programs to multiply matrices via the subblock technique and row technique were constructed and run on the Caltech/JPL Mark II hypercubes. To estimate efficiencies reliably,

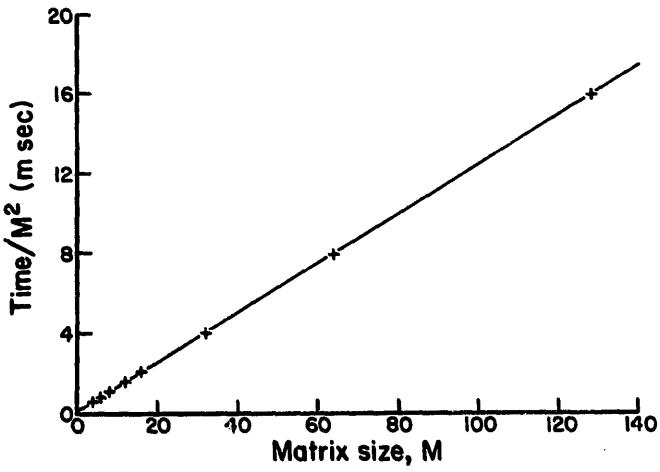


Fig. 4. Matrix multiplication timings for a single node processor running the usual, simple, sequential algorithm. The time divided by M^2 is actually plotted for clarity. Also shown is the fit giving Equation (16).

a third program was constructed which merely ran on a single node processor and multiplied matrices via the usual sequential machine algorithm. This gave us the 'time on single processor' needed in (13). All programs were written in the programming language 'C' and used the fast message passing facilities provided by the Crystalline Operating System, developed at Caltech [4,5]. All three programs were coded to give maximum performance for each. Indexing overheads and such were reduced to a minimum in each case so as to give a fair comparison of the algorithms.

In Appendix B, we summarize the relevant performance information for the Caltech/JPL Hypercubes, including a study of the values of t_{comm} and t_{flop}. Further information may be found in [12].

Timings for the sequential algorithm running on a single node processor are shown in Fig. 4. The times are fit very well by

$$T_{\text{single node}} = 119\ \mu\text{s}M^2 + 123\ \mu\text{s}M^3 \tag{16}$$

for $M \times M$ matrix multiplication (32 bit arithmetic throughout). The quadratic term (which appears in Fig. 4 as the very small, but non-zero, y-intercept represents the time taken to initialize the matrix C to zero at the beginning of the multiplication and the time to do some simple index arithmetic. As much index arithmetic as possible was pulled out of the two innermost loops of the program (the quadratic and cubic terms in (16)). The cubic term in (16) represents a floating-point speed of a single, 5 Mhz, 8086–8087 processor as 1/61.5 Mflops. This somewhat disappointing speed is consistent with other timings (given in Appendix B). This is more or less a worst-case result, however. As explained in Appendix B, the 'Mfloppage' of the processor increases (by a factor of about 2) for more complex expressions since the compiler is able to store intermediate results in 8087 registers.

Figure 5 shows timing results for a hypercube mapped to a 4×4 mesh and running the square subblock algorithm. What is actually plotted in Fig. 5 is T/M^2 versus M, so the asymptotic floating-point speed (the cubic term of (5)) shows as a linearly rising term, communications overhead shows as a constant (non-zero y intercept), and the pipeline startup time of the algorithm shows as a $1/M^2$ rising term at small M. It is quite clear from the figure that this implementation of matrix multiplication is quite efficient. Even for the modest matrix sizes such as 16×16 or 32×32 (total matrix size) on a 4×4 machine, the pipeline startup times and communication overheads are small effects. A linear fit to the points representing timings for the larger matrix sizes is also shown in Fig. 4. The slope coming from this is 7.64 μs, almost exactly 1/16 of 123 μs (see Equation (16)), as it should be.

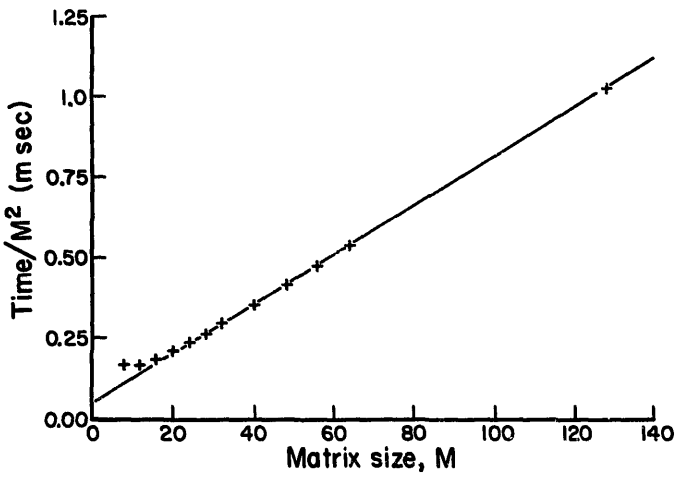


Fig. 5. Timings for the square subblock algorithm on a 4×4 machine. Again, time divided by M^2 is plotted.

An alternative way of presenting the 4×4 timing data is given in Fig. 6. Here, we have plotted $1/\epsilon - 1$ versus $1/\sqrt{n}$, with ϵ calculated directly from (13) and our single node timings. This is done so as to expose clearly the overhead terms of (14). The linear behavior for small $1/\sqrt{n}$ (large matrices) shows that the functional form of (14) is correct. A linear fit gives, as the coefficient of $1/\sqrt{n}$,

$$\frac{t_{\text{comm}}}{t_{\text{flop}}} = 1.39, \tag{17}$$

a reasonable figure for the Crystalline Operating System. One must bear in mind, however, that some of what we are calling 'communications overhead' (quadratic, M^2, terms in the timing equation) are actually other overheads, such as index initialization, which are dependent upon how one actually writes the code. The point is that, if one tries to write fast code, one removes as much index arithmetic as possible from the innermost loop of the program. This strategy lowers the coefficient of the cubic term (giving a faster code) but also adds additional quadratic effects which have identical dependence on n and N as the basic communication overhead.

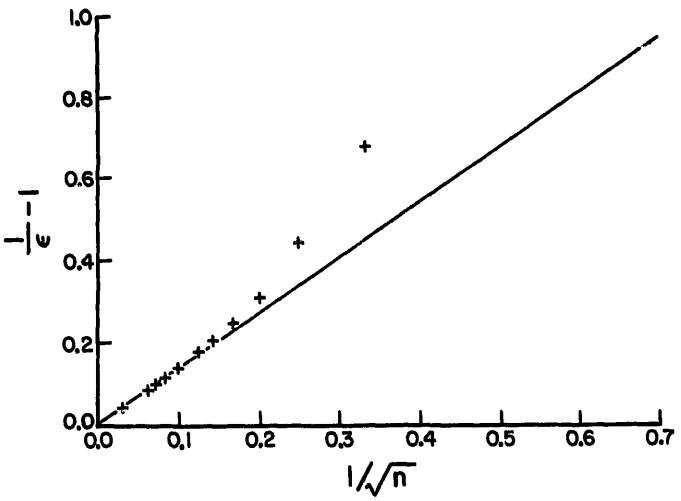


Fig. 6. The same timings as in Fig. 5, but plotted as overhead $(1/\epsilon - 1)$ versus $1/\sqrt{n}$ (n is the 'grain size', the number of matrix elements per node).

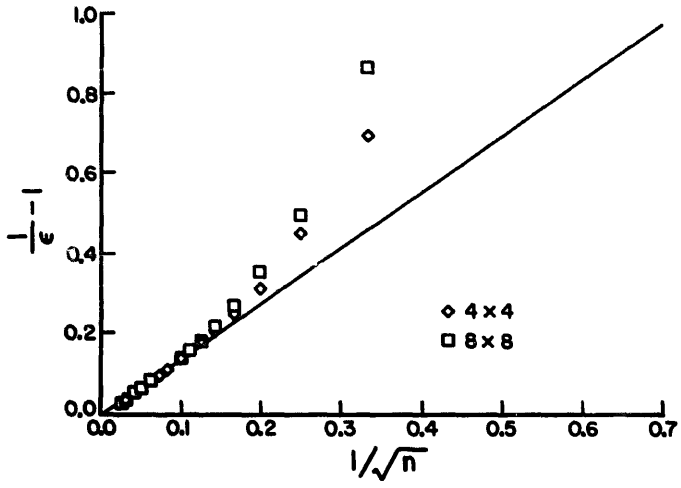


Fig. 7. The same as Fig. 6, but with timings from a 8×8 machine added, showing universal behavior.

To show that the result of (17) is 'universal', i.e., independent of the total machine size and dependent only upon the 'grain size', n, we offer Fig. 7. This is the same as Fig. 6, but with 8×8 machine timings added. As hoped, for $n > 100$ (a 10×10 submatrix in each node) the behavior is almost exactly universal. The deviation from this at small n is due to the larger pipeline startup time on the larger machine.

Let us now turn our attention to the 'pure row based' algorithm, or one-dimensional (ring) decomposition. This program was timed on the same machine sizes as before. Not surprisingly, there was no evidence of any pipeline startup effect as there was previously. Also as expected, the communication overhead for this algorithm is greater than for the subblock case. The timings are shown in Fig. 8, where the previous results of Fig. 7 are also shown for contrast. The strong non-universal behavior of the one-dimensional algorithm is quite apparent.

Fits give, as the coefficient of $1/\sqrt{n}$, 3.61 and 7.49 for the 16 and 64 node cases, respectively. First of all, the ratio of these slopes, 2.07, is within experimental error of 2, expected from (15). The ratio of 3.61 to 1.4, however, is 2.6, not very close to the factor of 2 expected from (14) and (15). The reason for this is, as discussed before, the effect of indexing masquerading as communication overheads. Due to the large aspect ratio, r, of the row

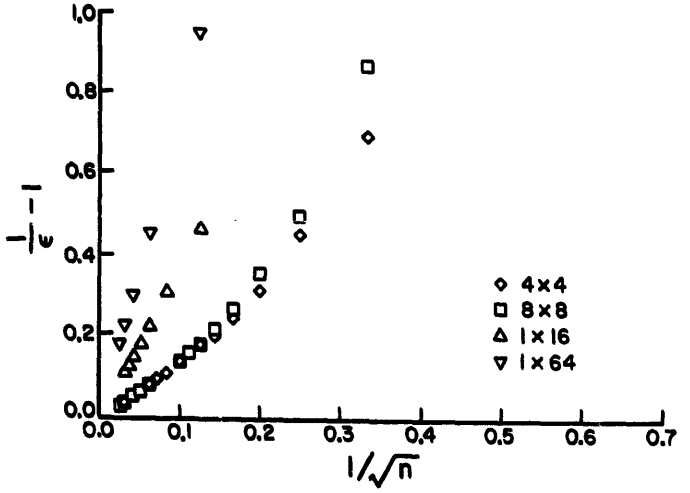


Fig. 8. The same as Fig. 7, but with timings from the one-dimensional algorithm added. The high overhead and strong non-universal behavior of this decomposition is quite apparent.

decomposition, it turns out that much more index initialization is required than for the square subblock case. This somewhat subtle point causes the row decomposition to have even more overhead than predicted by (15).

Actually, this is a very general phenomenon. Normally, a parallel algorithm executes the same total number of floating-point operations as the sequential algorithm. This is certainly the case for all the algorithms discussed here. Communication overheads and load imbalances degrade the performance from the perfect efficiency of 1. Another type of degradation, however, concerns the integer arithmetic associated with the indexing of the various arrays involved. This class of algorithm would use software whose inner loops were constructed as follows:

```
for (i running over m0 rows){
  Do some index arithmetic dependent upon decomposition
  to find associated columns for this processor.
  for (j running over m1 columns) {
    Inner loop identical to sequential code so that
    asymptotic efficiency is 1.}}
```

It usually happens (as it has here) that the parallel algorithm requires more indexing operations than the sequential algorithm, leading to another type of overhead, which we might term 'software decomposition overhead'. As indicated above, the software decomposition and communication overheads have similar dependencies on n and N. Though typically a small effect, it should be kept in mind when one is doing careful timings of parallel programs.

We have verified that the row decomposition is less efficient than the subblock and that the size of the overhead increases with machine size as $\sqrt{N}$, in accordance with (15).

5.2. *Padding for general sized matrix—Load imbalance*

Any practical matrix multiplication algorithm must be able to multiply matrices of arbitrary size—not just those whose order allows an exact mapping onto the hypercube with each processor having equal numbers of matrix elements. Towards this end, we have modified our algorithms so that they automatically pad the matrix up to the first size which matches the machine dimensions. The padding, of course, is done with 0's (1's on diagonal) so as not to effect the matrix product.

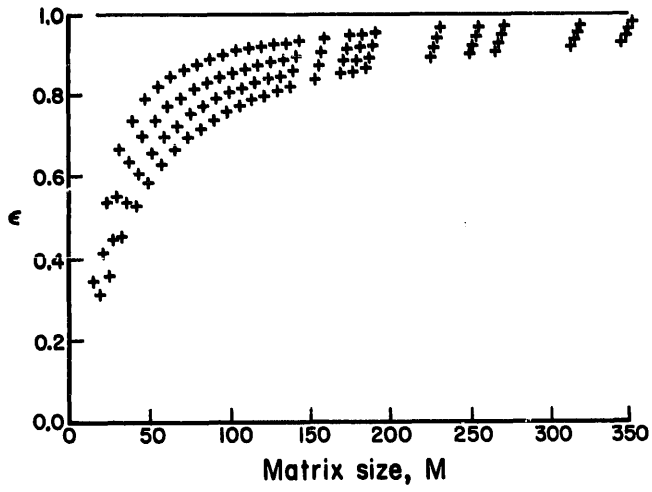


Fig. 9. Efficiency versus matrix size for an almost continuous set of matrix sizes, illustrating 'sawtooth' performance. This is for the square subblock algorithm on a 8×8 machine.

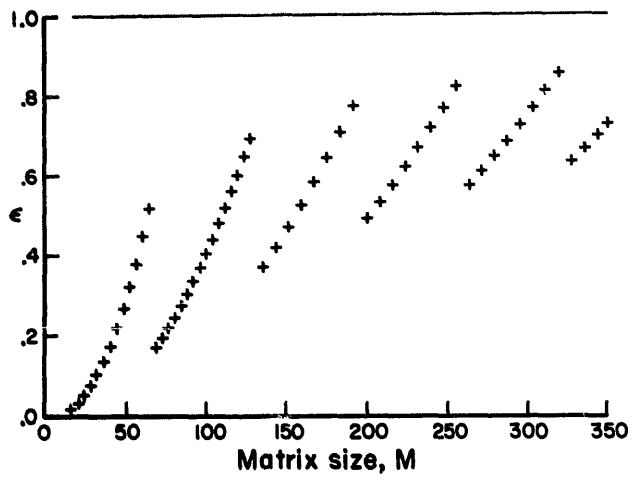


Fig. 10. Same as Fig. 9, but for the one-dimensional algorithm on a 1×64 machine.

Having done this, we present plots of efficiency, ϵ, versus M for almost a continuous range of M, for a 2^6 hypercube. The results appear in Figs. 9 and 10. Both algorithms display 'sawtooth' performance curves whose upper envelopes are described by (14) and (15). The striking fact, however, is that the severity of the 'sawtooth degradation' is much larger for the ring decomposition. The reason for this is illustrated by the following example. If one wishes to multiply 65×65 matrices on a 64 node ring, one must pad by 63 rows to a matrix size of 128. If this same multiplication is done on a 8×8 machine, however, the amount of padding necessary is only 7 rows, resulting in a 72×72 multiplications.

This sawtooth effect may be a much more important practical consideration governing the choice between the algorithms than (14) and (15).

6. Conclusions

We have shown how to implement efficiently matrix multiplication on a class of concurrent processors typified by the hypercube. Our theoretical performance analyses have been confirmed by explicit timing of implemented algorithms. The efficiency (speed up divided by the number of nodes) fits the general form

$$\epsilon \sim \frac{1}{1 - \frac{c}{\sqrt{n}} \frac{t_{\text{comm}}}{t_{\text{flop}}}} \tag{18}$$

discussed in [2] for a range of problems. The coefficient, c, equals 1 for the optimal square subblock decomposition. For the row decomposition the same coefficient is $\frac{1}{2}\sqrt{N}$ which grows as one increases the number of processors, N. Even for $N = 64$, the results presented here clearly demonstrate the inferiority of the row decomposition. We also noted that load imbalance (unequal sized regions in each processor) was far more severe in the pure row case.

We will find that these conclusions extend to the other matrix algorithms although the coefficient, c, differs from case to case. In defense of the pure row algorithm, it is true that it allows one to adapt standard sequential code to the concurrent processor more easily. We are not certain whether this is fundamental or a byproduct of the way standard languages (FORTRAN) store matrices. In any case, we feel that at this stage we wish to study the optimal algorithms—especially ones that scale properly to large numbers of nodes—rather than the algorithms that convert most rapidly to the parallel environment.

Appendix A. Pipes (broadcasts) on the hypercube

A fundamental operation used in the concurrent matrix multiplication is the broadcast of information among a set of processors. In general this is not to all nodes in the machine but just to those in a 'row of processors' in a two-dimensional mesh mapping of the hypercube. If one examines this map [14], it becomes clear that the nodes in a row (or column) of such a 2D map corresponds to a subcube of the original hypercube. For instance consider a 64 (2^6) node hypercube decomposed into a 8×8 two-dimensional mesh. Each of the eight 'rows' of processors corresponds to an eight node (2^3) hypercube; further, each of these subcubes is independent, i.e., message traffic on them does not interfere. Thus we find that the broadcasting step in the algorithm corresponds to finding the optimal way of broadcasting on a hypercube itself.

Consider the problem of broadcasting n words on a cube of dimension v (as described above, v is one half the full cube index for the square subblock decomposition while $n = m^2$ in the notation of Section 3. The optimal algorithm for this broadcast depends on details of the hardware. In fact, our newer hypercubes (the Mark III) under construction support arbitrary subcube broadcasts in the hardware [13], and there is a simple fast algorithm. However, the Mark II machines have no special support for either full cube or subcube broadcasts; these must be implemented in software. In the Mark II (see Appendix B), information is communicated in packets containing 8 bytes and so one needs to broadcast $p = \frac{1}{2}n$ packets.

In the case of a single packet, $p = 1$, the best hypercube broadcast algorithm corresponds to a well-known technique that maps the cube into binary tree. This is illustrated in Fig. A.1. and

$$t_{\text{broadcast}}(p = 1) = v(2t_{\text{comm}}) \tag{A.1}$$

where the factor of 2 in (A.1) corresponds to our conventional choice that t_{comm} refers to the time to communicate a single word. Equation (A.1) can be generalized to arbitrary p, to find

$$t_{\text{broadcast}}(p) = (p + 1)vt_{\text{comm}}. \tag{A.2}$$

However, although the tree algorithm is the fastest way of rolling a packet through the cube, one must send out the packet on each of v channels on the initial node. For this reason, the time in (A.2) grows like vp.

There is clearly an alternative technique which is to ignore the full hypercube connectivity, and just pipe packets along the row of processors regarded as having a periodic ring topology. This is illustrated for the case $v = 3$ in Fig. A.2 for the optimal split ring algorithm. This can be shown to take a time

$$t_{\text{broadcast}}(p) = (2^v + 2p - 2)t_{\text{comm}}. \tag{A.3}$$

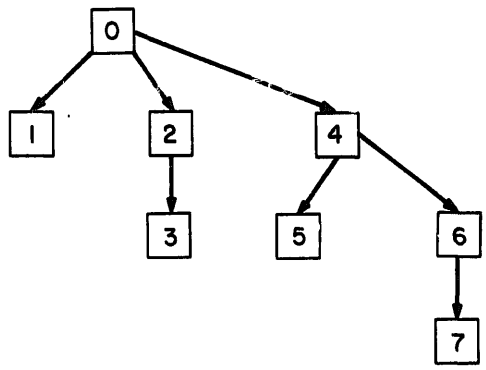


Fig. A.1. The 'subcube broadcast' algorithm illustrated for a 3 cube.

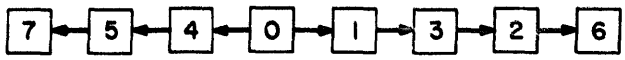


Fig. A.2. The 'split ring' algorithm illustrated for a 3 cube.

Comparing (A.2) and (A.3), we see that (A.3) has a smaller $2 < \nu$) coefficient of p but a larger start up time (2^ν versus ν). For the large values of $p \sim \frac{1}{2}m^2$ needed in our algorithm, (A.3) is always faster than (A.2).

One can improve Eq. (A.3) in ways that are quite complicated and we have not implemented as they represent unimportant speed improvements. A reasonably simple step is to use a 'homogeneous split-ring' where the splitting point (between node 6 and 7 in Fig. A.2) is chosen uniformity throughout the ring. It can be seen that this changes the coefficient of p in (A.3) from 2 t_{comm} to $2(1-2^{-\nu})$ t_{comm}. One can further speed up the algorithm and reduce the startup time in (A.3) by using composite algorithms which combine features of those in Figs. A.1 and A.2. We are able to derive these in special cases (low values of p and ν) but unable to find a general expression for the optimal algorithm.

Appendix B. Performance of the Caltech Mark II hypercube

The hardware used is described in [16], and we will only give a brief description here. Each node is a single board computer built around the INTEL 8088-8087 microprocessors; at the time the data here was taken, the boards ran at a 5 Mhz clock (we are upgrading to 8 Mhz). Each node has $\frac{1}{4}$ Mbytes of memory and a 2^d node system uses d channels on each board (out of a maximum of 8) to communicate with its neighbors in the hypercube architecture.

[12] represents a detailed discussion of the basic performance of the calculation (8087-8087) and communication sections of the system. We use the parameters t_{comm} and t_{flop} introduced in Section 3.2 (and [12]) to describe the performance of the machine. These are respectively the fundamental communication and calculation times for 4 byte (single word) operations.

The floating-point performance of the node boards was measured to be in the range

$$t_{\text{flop}} = 34\,\mu\text{s} \rightarrow 57\,\mu\text{s}. \tag{B.1}$$

These numbers need to be increased by 10 to 30% for index and loop overheads in realistic codes. The lower number in (B.1) corresponds to a relatively complex calculation—explicitly, $a = b \cdot c + c \cdot d + d \cdot e$, where a, b, c, d, e are floating-point numbers—where one can make use of the internal registers of the 8087. The higher number in (B.1) corresponds to 'bare bones' calculations—explicitly, $a = b \cdot c$, where the overhead of loading the 8087 in 16 bit pieces is higher. As shown in Section 5.1 we see a performance that corresponds to a t_{flop} at the upper end of the range (B.1).

The code used in our implementations uses the so-called "Crystalline Operating System"or CrOS for its communication [4]. This software allows the user to send to different nodes and receive messages from them. The application in this paper only uses message passing between nodes that are directly connected in the hypercube topology. This obviates any need for message forwarding and allows a simple, fast operating system. The messages are bundled into 8 byte packets corresponding to the size of the FIFO's used to store messages on each channel. The value of t_{comm} for CrOS depends somewhat on the number of packets transferred and in Fig. B.1, we plot t_{1}comm versus the message size and compare it with the value of t_{flop} given in

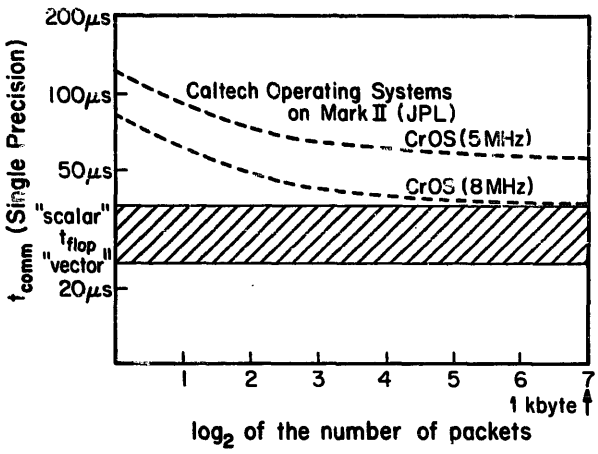


Fig. B.1. Plot of t_{comm} versus the number of 64 bit packets in the message, for the Crystalline Operating System (CROS) running on the Caltech/JPL Mark II Hypercube. "Single Precision" refers to the fact that t_{comm} is defined to be the time to transfer a single precision (32 bit) datum. Also shown is t_{flop}, the time to do a floating-point operation (32 bit). This is plotted as a band, with extremes of 'scalar' and 'vector', since the speed of the 8086–8087 processor in floating-point arithmetic is a strong function of the complexity of the expression being evaluated. See [12] for further details.

(B.1). One is typically transferring several packets at a time (m^2 or m_1^2 words in notation of text with 2 words per packet) and for this case, one finds a communication time $t_{comm} \sim 70\,\mu s$.

References

[1] E. Brooks et al., Pure gauge SU(3) lattice theory on an array of computers, *Phys. Rev. Lett.* **52** (1984) 2324.

[2] G. Fox, Matrix operations on the homogeneous machine, Unpublished Caltech Report Hm-5 (CALT-68-939), 1982.

[3] G. Fox, The performance of the Caltech hypercube in scientific calculations. *Proc. Symposium on Algorithms, Architectures and the Future of Scientific Computation* (1985).

[4] G. Fox, editor, Caltech JPL concurrent computation project—Annual report 1983–1984 and recent documentation Volume 1: Tutorial and system documentation, Unpublished Caltech Report, 1985.

[5] G. Fox, G. Lyzenga, D. Rogstad and S. Otto, The Caltech concurrent computation program—Project description, *Proc. 1985 ASME International Computers in Engineering Conference* (1985).

[6] G. Fox and S. Otto, Algorithms for concurrent processors, *Phys. Today* (May 1984).

[7] G.A. Geist and M.T. Heath, Parallel Cholesky factorization on a hypercube multiprocessor, ORNL-6190 Technical Report, 1985.

[8] M.T. Heath, Parallel Cholesky factorization in message-passing multiprocessor environments, ORNL-6150, Oak Ridge National Laboratory, Technical Report, 1985.

[9] L. Johnsson, Dense matrix operations on a torus and a boolean cube, prepared for *National Computer Conference*, July 15–18, 1985 in Chicago, IL.

[10] H.T. Kung and C.E. Lieserson, Algorithms for VLSI processor arrays, in: C.A. Mead and L.A. Conway, eds., *Introduction to VLSI Systems* (Addison-Wesley, Reading, MA, 1980).

[11] C. Moller, Presentation at *Knoxville Hypercube Meeting*, August 26–27, 1985.

[12] S. Otto, A. Kolawa and A. Hey, Performance of the Mark II hypercube, Unpublished Caltech Report Hm-188, 1985.

[13] J.C. Peterson, J. Tuazón, D. Lieberman and M. Pniel, The Mark III hypercube—Ensemble concurrent computer *Proc. 1985 International Conference on Parallel Processing* (1985).

[14] J. Salmon, A program to automatically decompose a physical lattice into a hypercube, Unpublished Caltech Technical Report Hm-53, 1984.

[15] C. Seitz, The cosmic cube, Comm. *ACM* **28** (1985) 1.

[16] J. Tuazon, J. Peterson, M. Pniel and D. Lieberman, Caltech/JPL hypercube concurrent processor, *Proc. 1985 International Conference on Parallel Processing* (1985).

[17] S. Utku, M. Salama and R. Melosh, Concurrent Cholesky factorization of positive definite banded hermitean matrices, Unpublished Caltech Report Hm-106, 1984.