Toward Scalable Algorithms for Orthogonal Shared-Memory Parallel Computers

Isaac D. Scherson, Ashish Mehra and Jennifer L. Rexford

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

Abstract

We consider the problem of developing scalable and near-optimal algorithms for orthogonal shared-memory multiprocessing systems with a multi-dimensional access (MDA) memory array. An orthogonal shared memory system consists of 2^n processors and 2^m memory modules accessed in any one of m possible access modes. Data stored in memory modules are available to processors under a mapping rule which allows conflict-free data reads and writes for any given access mode. Scalable algorithms are presented for two well known computational problems, namely, Matrix Multiplication and the Fast Fourier Transform(FFT). A complete analysis of the algorithms based on computational time and the access modes needed is also presented. The algorithms presented scale very well onto higher dimensional MDA architectures but are not always optimal. This reveals an existing trade-off between the scalability of an algorithm and its optimality in the MDA computational model.

1 Introduction

Much attention has been focused on orthogonal shared memory multiprocessing architectures like the OMP [6,9] and the RMOT [1]. Algorithms have been mapped successfully onto these architectures showing their potential application in image processing, graph problems, vector computations and sorting. Since the introduction of OMP, Important generalizations were suggested by Hwang [5] and by Scherson [10,11]. The work in [8,10,11] is based on an elegant definition of orthogonal graphs which leads to the construction of several hypercube-like machines. An orthogonal graph is defined as a set of 2^m nodes, which in turn are linked by 2^{m-n} edges for every link mode q defined in an integer set $Q* = \{q_i | 0 \leq q_i \leq m-1\}$. Given m-bit node labels, a mode-q-link exists between two nodes if and only if the labels match over n bits starting at bit position $q \in Q*$. These orthogonal graphs lead to a number of interesting parallel computer structures. Distributed systems with a message passing network whose topology is defined by an orthogonal graph is an obvious example. Multistage interconnection networks characterized by orthogonal graphs are reported in [11].

[0]This work was supported in part by The Air Force Office of Scientific Research under grant number AFOSR-90-0144.

Orthogonal shared memory systems are obtained by associating a memory module with each graph node. The 2^m-element memory array is then accessed by 2^n processing elements according to the graph construction rule. That is, a processor $P_{\underline{x}}$ accesses a memory module $M_{\underline{y}}$ under some mode $q \in Q*$ if and only if the n bits of the m-bit index $\underline{y}$, starting at bit position q, are equal to the n-bit index $\underline{x}$. A number of well known machine architectures exhibit the shared memory topology which results from the application of the orthogonal construction rule. OMP is only one example. In addition, STARAN, with its Multi-Dimensional Access (MDA) memory [2], also falls in the category of shared memory orthogonal structures. Figures 1 and 2 illustrate an MDA system with 64 memory modules and 8 and 16 processing elements respectively. The boxes represent memory modules and are assumed to be numbered in the natural row-major order. The number in each box represents the index of the processor which can access that memory module for the given access mode.

The notation in this paper is based on [8]. Architectures are referred to as $MDA(n, m, \{q_0, q_1, ..., q_l\})$, for some integer $l < m$. The system is connected if and only if

$$\forall i \in [0, m-1],\ \exists q \in Q* \ni |(q+n) \bmod m - i| < (m-n)$$

In general, an OMP is given by $MDA(m/2, m, \{0, m/2\})$, while STARAN's memory is $MDA(m/2, m, \{0, 1, ..., m-1\})$. The shared memory equivalent of the binary hypercube is $MDA(m-1, m, \{0, 1, .., m-1\})$. A very useful class of orthogonal graphs is one in which $Q*$ is a set of disjoint modes. These graphs are characterized by the fact that in the connectivity expression above, for each $i \in [0, m-1]$ there exists a *unique* $q \in Q*$ which satisfies the condition. The minimum number of access modes to guarantee conectivity in a graph with disjoint modes is $\omega = \frac{m}{m-n}$. Such orthogonal graphs were named *Omega Graphs* and can be used effectively to define spanning bus arrays of loosely or tightly coupled parallel processing systems.

In light of this generalized parallel computing structure, several issues of algorithm-mapping assume importance. One important problem is to devise techniques which allow scaling of a problem to parallel systems which differ in the number of resources. We tackle here the scalability problem for orthogonal shared memory systems. We assume that a fixed size problem (thus m is a constant) is to be solved in MDA

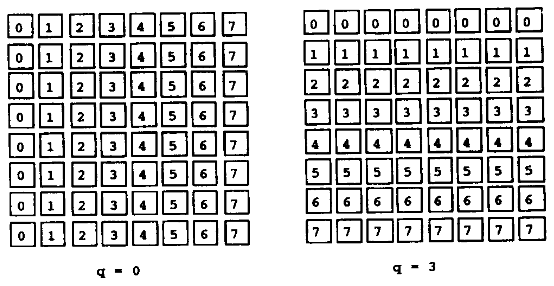


Figure 1: MDA(3,6,{0,3})

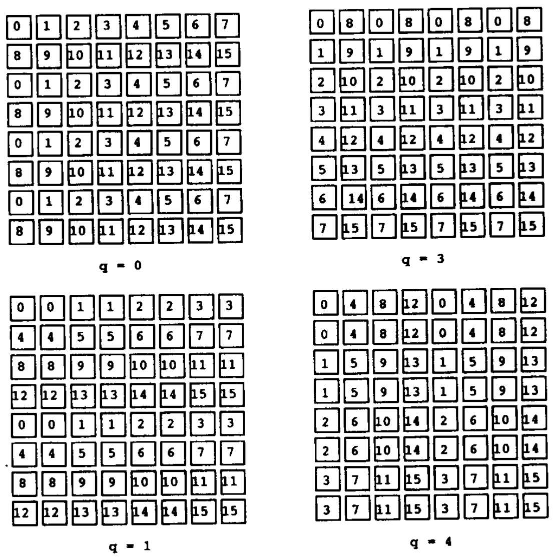


Figure 2: MDA(4,6,{0,1,3,4})

systems where n and $Q*$ may vary. That is, we allow changes in the number of processors and the memory to processor interconnect. Because of the large variety of systems which can be described with orthogonal graphs, scalability of the algorithms becomes important not only because it simplifies matters but also because it is a very useful concept towards making algorithms independent of the particular parallel computing structure. As will be seen later, scalability does not always ensure optimality. This trade-off between scalability and optimality can be best understood by considering practical problems. In this paper, we explore these issues by looking at two very well known and fundamentally important computational problems, namely Matrix Multiplication and the Fast Fourier Transform (FFT). An analysis of each algorithm is presented to determine their computational times and the number and type of access modes needed (hence required connectivity).

2 Matrix Multiplication

In [6] and [9] algorithms are presented for matrix multiplication in the OMP. However, we have found that the algorithms, although optimal, do not extend to higher dimensional MDAs. A different approach needs to be taken. As we speculate that it is easier to scale down than to scale up, we consider algorithms which will work on the highest dimensional orthogonal structure with all possible access modes, that is $MDA(m-1, m, \{0, 1, ..., m-1\})$. From it, as n decreases from $m-1$ to 0 (one processing element), access modes in $Q*$ are deleted to yield the minimum MDA system capable of executing the procedure. This is an interesting observation as it points to the development of algorithms for the largest possible system and scaling down to fit any other system with the desired number of processing resources.

2.1 Suggested Algorithm

Consider the multiplication of two $r \times r$ matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ where $r = 2^p$. The resulting product matrix is denoted by $C = [c_{ij}]$. In an MDA system with 2^m memory modules, the matrices A and B are assumed to be stored in row major form such that a_{ij} and b_{ij} are stored in a memory module whose index is obtained by concatenating the binary expansions of the indexes i and j. We shall hence refer to such memory module as M_{ij}. Clearly $p = m/2$. An algorithm is proposed below and claimed to be correct independently of the value chosen for the number of processor in the system (choice of n). For any given n, the set $Q*$ must be determined such as to provide connectivity in the resulting orthogonal system.

The generalized algorithm for matrix multiplication is based on a well known procedure for hypercube machines [4] and comprises the following steps:

1. ***Broadcast*** the rows of A and the columns of B to the diagonal memory modules.

   Either the forward and/or backward diagonals need to be used. Because connectivity depends on the choice of n, access modes must be chosen to guarantee connectivity and hence all processors may access simultaneously either the main and/or the backward diagonal of the memory array. The rows in the top half of A are sent to the forward diagonal while the rows in the bottom half are sent to the back diagonal. The columns of B are sent to both diagonals.

2. *Align* the column elements with the row elements for correct computation.

   This step is needed since the row and column vectors may not be directed identically in the back-diagonal memory modules and hence they need to be aligned properly.

3. *Repeat until done*

(a) *Multiply*: Each of the processors multiplies the two operands available in the diagonal memory module accessible to it and places the result back in the same memory module.

(b) *Route* the diagonal elements back to the rows.

(c) *Add* the components of the elements under suitable access modes and place the final result accordingly.

loop

Data is distributed among the memory modules such that the elements a_{ij} and b_{ij} are in memory module M_{ij}. A suitable choice of access modes is essential to accomplish the three main steps in the algorithm. Access modes must be such that every row and every column can be sent to the diagonals in a finite number of steps. This is ensured by guaranteeing connectivity and all other operations can be accomplished using the same access modes. Routing back to the rows is the exact reverse of broadcasting and utilizes the access modes in the reverse order. It turns out that the modes needed for addition are the same as the modes needed for the column broadcast.

The elements of the matrix C are defined by the following equation:

$$c_{ij} = \sum_{k=0}^{2^p-1} a_{ik} * b_{kj} \quad i,j = 0,1,..,2^p - 1$$

To compute element c_{ij}, we require row i of A and column j of B. Since all rows and columns are sent to the diagonal memory modules (broadcasting step), row i of A and column j of B are present in the same diagonal and a_{ik} and b_{kj} are in the same memory module. Processors having access to these diagonal memory modules compute the required partial products which form the components of the dot product for element c_{ij}. These individual products are then moved back to row i for $i = 0,1,..,2^p - 1$, and summed up in suitable access modes thereby giving rise to 2^p elements of the matrix C. This is done for $j = 0,1,..,2^p - 1$ to generate any c_{ij}. Thus, all elements of the matrix C can be computed and the algorithm is correct.

2.2 Analysis of the algorithm

The implementation of matrix multiplication on MDA architectures presents some interesting problems in light of the suggested algorithm. To have a scalable algorithm, we need to work through the diagonal memory modules. Omega graphs provide connectivity and hence all steps in the algorithm can be carried out. However, because of the access patterns of some omega graphs, broadcasting may become a nearly sequential process. To optimize the utilization of the concurrency available in the algorithm, additional access modes could be added. Thus, the choice of access modes for a given n (number of processors) impinges on the utilization of the processing resources.

Consider an omega graph with 2^m nodes. Following [10] again, this as a spanning bus structure with 2^{m-n} nodes on each spanning bus for each dimension q defined in the integer set $Q*$. We fix m and vary n, therefore we are scaling the architecture from the OMP on one end to the hypercube on the other. The OMP has $2^{m/2}$ nodes in each dimension whereas the hypercube has 2. Thus, other intermediate architectures should have $2^{m/2-t}$ nodes on each spanning bus, assuming a linear change of the exponent. The variable t takes on the values $t = 0,1,...,(m/2 - 1)$. Note that this defines a class of graphs different from omega graphs, since omega graphs are characterized by $2^{m/\omega}$ nodes in each dimension. As the total number of memory modules in the system is fixed, the following must hold true:

$$2^{\frac{m}{2}-t} \times 2^n = 2^m$$

This equation leads to the relation $m = 2n - 2t$, where $t = 0,1,..,(m/2 - 1)$. Define $k = 2t + 2$. The relation now becomes $m = 2n - (k-2)$, where $k = 2,4,6,..,m$. All MDA architectures can now be specified by this generating equation. k is an even integer and the maximum value it can take is m. It determines the number and type of access modes needed and also specifies the communications overhead [9,12]. Note that not all available modes may be needed. There are m possible access modes but only some are useful for achieving the desired connectivity in the network. Let us illustrate this by a complete analysis of the algorithm for $MDA(n,m,\{0,1,..,m-1\})$:

- *$m = 2n - (k-2)$ $\quad k = 2,4,6,..,m$*
- *Total number of modes needed = k*
- *Modes needed for the broadcast of the rows of A and for routing:*
  For $k = 2$, $q = 0$
  For $k > 2$, $q = 0, m/2+1, m/2+2, ..., m/2+(k/2-1)$
- *Modes needed for the broadcast of the columns of B and for addition:*
  For $k = 2$, $q = m/2$
  For $k > 2$, $q = m/2, 1, 2, ..., (k/2-1)$
- *Number of steps needed for broadcast/routing = $k/2$*
  In these many steps, $2^{n-m/2}$ rows/columns are moved to the diagonal(s) or back. Therefore, the overhead is $O(k/2^{k/2})$ steps per row/column in broadcasting and routing. For addition, the overhead can be made $O(k)$ by computing the partial products for all the elements before starting the additions. In this manner, all the processors can be made to work in the addition stage.
- *Multiplication is done by $2^{m/2+1}$ processors since there are only two diagonals. The other processors are masked out in this stage. Multiplication proceeds in parallel in the two diagonals. Hence, it takes $O(2^{m-1})$ steps.*
- *$k = 2$ gives the well known OMP architecture. Here, only 2 modes are needed. Note that $k = 2$ implies $m = 2n$.*
- *$k = m$ generates the hypercube. Here, all the m modes are necessary. Note that $k = m$ implies $m - n = 1$, the relation for the hypercube architecture as shown in [8].*
- *It must be noted here that both m and k are even integers.*

As has been mentioned before, the processor utilization can be increased by doing as much as possible in a particular mode before changing the mode. This is true for broadcasting/routing as well as addition. Clearly, the above generalization simplifies the analysis of the algorithm for a given architecture. This probably has some significance in the light of a scaling in the number of processors in the system. More *degrees of freedom,* so to say, are needed in systems with a larger number of processors and for the same number of memory modules.

2.3 Case Study

Case A: Consider $MDA(n, m, \{0, 1, .., m-1\})$ with the restriction $m = 2n - 2$. Evidently, this is the case with $k = 4$. As will be seen later, the only modes used are $q = 0, 1, m/2, m/2 + 1$. This architecture generates the hypercube for $n = 3, m = 4$.

1. *Broadcast* rows of A and columns of B to the forward and back diagonals:

   (a) *Broadcasting of rows of A:* Move the top half rows of A to the forward diagonal.

   Starting mode: $q = 0$

```
i ← 0
while $i \leq 2^{p-1} - 2$
do
    For k=i & i+1 in parallel do
        For $j=0$ to $2^p - 1$ in parallel do
            $M_{kj}(a_{kj}) \rightarrow M_{jj}$ in 2 steps,
             under modes q=0,m/2+1
        loop
i ← i + 2
loop
```

   (b) *Broadcasting the rows of A:* Move the lower half rows of A to the back diagonal.

```
while $i \leq 2^p - 2$
do
    For k=i & i+1 in parallel do
        For $j=0$ to $2^p - 1$ in parallel do
            $M_{kj}(a_{kj}) \rightarrow M_{(r-1-j)j}$ in 2 steps,
             under the modes q=0,m/2+1
        loop
i ← i + 2
loop
```

   (c) *Broadcasting of columns of B:* Move each column to both diagonals.

   Starting mode: $q = m/2$

```
j ← 0
while $j \leq r - 2$
do
    For k=j & j+1 in parallel do
        For $i=0$ to $2^p - 1$ in parallel do
            $M_{ik}(b_{ik}) \rightarrow M_{ii}$
            $M_{ik}(b_{ik}) \rightarrow M_{i(r-1-i)}$
        (Each is performed in 2 steps,
            under modes q=m/2 and q=1 respectively)
        loop
j ← j + 2
loop
```

   (Let s be the column index within the diagonal memory modules and l be the row index within the diagonal memory modules)

2. *Align* the columns of B with the rows of A (for correct computation) in the diagonal memory modules.

```
For s=0 to $2^p - 1$
    For i=0 to $2^p - 1$
        $M^s_{i(r-1-i)} \rightarrow M^s_{(r-1-i)i}$
    loop
loop
```

   Each transfer is done in two steps and under the modes $q = m/2$ and $q = 0$ respectively. Data is read in mode $q = m/2$ from the backward diagonal memory modules and written back onto these very memory modules under the mode $q = 0$. This transfer is done by the processors having back diagonal memory access under the modes $q = 0$ and $q = m/2$.

3. *Computation:*

   Starting mode: $q = 0$

```
For $s = 0$ to $2^p - 1$
    (a) For l=0 to $2^{p-1} - 1$ do
        Multiply:
            For i= 0,1,,,$2^{p-1}$ in parallel do
            Forward diagonal: $M_{ii} \leftarrow M^l_{ii} * M^s_{ii}$
            Backward diagonal:
                $M_{i(r-1-i)} \leftarrow M^l_{i(r-1-i)} * M^s_{i(r-1-i)}$
            (These steps are computed in parallel $\Rightarrow$ 1step)
        loop

    This loop for l amounts to computing $a_{ij} * b_{ij} \rightarrow c_{ij}$,
    $i = 0, 1 \ldots r-1; j = s$
    for $2^p$ elements(each column) of the product matrix C

    (b) Route back to the rows:
        For l=0 to $2^{p-1}$
            For j=0,1,..,$2^{p-1}$ in parallel do
            Forward diagonal routing: $M_{lj} \leftarrow M^l_{jj}$
            Backward diagonal routing:
                $M_{(l+2^{p-1})j} \leftarrow M^l_{(r-1-j)j}$
            (These steps are computed in parallel)
        loop

        Each transfer is done in 2 steps under modes $q = 0$ and
        $q = m/2 + 1$ respectively

    (c) Add the components of each element placed in the rows,
        in the modes $q = m/2$ and $q = 1$ and place the result
        in the memory module $M_{is}$,  $i = 0, 1, \ldots 2^p - 1$
        i.e For $i = 0, 1, \ldots, 2^p - 1$ in parallel do
            $M_{is} \leftarrow \sum_{j=0}^{2^p-1} M_{ij}$ done in modes q=m/2,1
loop
```

Case B: Now consider $MDA(n, m, \{0, 1, \ldots, m-1\})$ with $m = 2n$. Clearly, this case corresponds to $k = 2$. This is very similar to the previous algorithm. However there are some differences as can be seen. The only modes needed here are $q = 0$ and $q = m/2$.

1. *Broadcast* rows of A and columns of B to the forward and back diagonals:

   (a) *Broadcasting of rows of A:* Move the top half rows of A to the forward diagonal and the bottom half to the back diagonal.
   Starting mode: $q = 0$

```
i ← 0
while i ≤ $2^{p-1} - 1$
do
    For j=0 to $2^p - 1$ in parallel do
        $PE_j : M_{ij}(a_{ij}) \rightarrow M_{jj}$
            in 1 step, under mode q=0
        $PE_j : M_{(i+2^{p-1})j}(a_{(i+2^{p-1})j}) \rightarrow M_{(r-1-j)j}$
            in 1 step under mode q=0
    loop
i ← i + 1
loop
```

   (b) *Broadcasting of columns of B:* Move each column to both diagonals.
   Starting mode: $q = m/2$

```
j ← 0
while j ≤ $2^p - 1$
do
    For i=0 to $2^p - 1$ in parallel do
        $M_{ij}(b_{ij}) \rightarrow M_{ii}$
        $M_{ij}(b_{ij}) \rightarrow M_{i(r-1-i)}$
    loop
j ← j + 2
loop
```

   (Let s be the column index within the diagonal memory modules and l be the row index within the diagonal memory modules)

2. *Align* the columns of B with the rows of A (for correct computation) in the diagonal memory modules.

```
For s=0 to $2^p - 1$
    For i=0 to $2^p - 1$
        $PE_i : M^s_{i(r-1-i)} \rightarrow M^s_{(r-1-i)i}$
    loop
loop
```

   Each transfer is done in two steps and under the modes $q = m/2$ and $q = 0$ respectively. Data is read in mode $q = m/2$ from the back diagonal memory modules and written back under the mode $q = 0$.

3. *Computation:*
   Starting mode: $q = 0$

```
For s = 0 to $2^p - 1$
    (a) For l=0 to $2^{p-1} - 1$ do
        Multiply:
            For i= 0,1,..,$2^p - 1$ in parallel do
            Forward diagonal: $M_{ii} \leftarrow M^l_{ii} * M^s_{ii}$
            Backward diagonal:
                $M_{i(r-1-i)} \leftarrow M^l_{i(r-1-i)} * M^s_{i(r-1-i)}$
            (These steps are computed sequentially
              and hence take 2 steps)
        loop

    (b) Route back to the rows:
        For l=0 to $2^{p-1} - 1$
            For j=0,1,..,$2^p - 1$ in parallel do
            Forward diagonal routing: $M_{lj} \leftarrow M^l_{jj}$
            Backward diagonal routing:
                $M_{(l+2^{p-1})j} \leftarrow M^l_{(r-1-j)j}$
            (These steps are computed sequentially)
        loop

        Each transfer is done in 1 step under mode $q = 0$

    (c) Add the components of each element placed in the rows,
        in the mode $q = m/2$ and place the result in the
        memory module $M_{is}$  $i = 0, 1, \ldots 2^p - 1$
        i.e For i=0,1,...$2^p - 1$ in parallel do
            $M_{is} \leftarrow \sum_{j=0}^{2^p-1} M_{ij}$, done in mode q=m/2
loop
```

2.4 Comments

The essence of the algorithm given lies in the broadcast of the rows and the columns to the diagonals. Once the rows and the columns have been sent to the diagonals, the operands are multiplied for each column and all the rows in parallel, the products routed to the row-wise memory modules, and added under suitable modes thereby yielding 2^p elements for each column. This is repeated for each column until all the elements are computed.

The same algorithm maps well onto all MDA architectures, including the OMP and the hypercube. However, for the $MDA(n, 2n, \{0, n\})$, which is nothing but the OMP, the back diagonal need not be used. Since the number of processors is equal to the number of forward diagonal memory modules, no speed-up can be obtained by using the other diagonal. Hence, the algorithm for $MDA(n, 2n, \{0, n\})$ can be implemented using just the forward diagonal yielding the following modified algorithm:

```
For k=0 to $2^p - 1$
    For m=0,1,..,$2^p - 1$ in parallel do
        Broadcast⇒ $PE_m : a_{km} \rightarrow M_{mm}$
                (The transfer is done in Column Mode, q=0)
                ($a_{km}$ is contained in memory module $M_{km}$)
    For j=0 to $2^p - 1$
        For m=0,1,..,$2^p - 1$ in parallel do
            Compute: Multiply
                $PE_m : a_{km} * b_{mj} \rightarrow c_{mj}$ (Row Mode, q=n)
                ($a_{km}$ is in $M_{mm}$, $b_{mj}$ is in $M_{mj}$, $c_{mj}$ is in $M_{mj}$)
    loop
    Add in Column Mode (q=0) and place elements in
    $M_{kj}$,j=0,1,..,$2^p - 1$ i.e.,
        For j=0,1,..,$2^p - 1$ in parallel do
        $PE_j : M_{kj} \leftarrow \sum_{i=0}^{2^p-1} M_{ij}$, in mode q=0
loop
```

The difference here is that now the elements are computed row-wise whereas, previously, they were being computed column-wise. In this improved algorithm, there is no aligning overhead since the back diagonal is not used at all. Also, for the same reason, there is no column broadcasting overhead. Such simplifications and improvements are, however, not possible for other MDA architectures because of the type of memory accesses provided. One can, of course, find another algorithm which does not involve any broadcast to the diagonal modules but which requires internal register memory at each PE and shifting of some columns (which cannot be accessed otherwise), and hence would incur some overhead. Most importantly, one then fails to see the correspondence between the various orthogonal shared memory multiprocessing systems. It must be mentioned here that for the aligning step in the algorithm presented here, internal register memory is assumed for processors accessing the back diagonal memory modules under modes $q = 0$ and $q = m/2$. Nowhere else is any memory needed or used. For better utilization of processors, one can modify the algorithm to finish all the partial multiplications before routing and then add the components together. This way all the processors can be made to work in the addition stage.

2.5 Complexity Analysis for Case Study

Recall that all matrices are $r \times r$ where $r = 2^p$, $p = m/2$. Consider $MDA(n, m, \{0, 1, m/2, m/2+1\})$. Here $m = 2n-2$ and hence $k = 4$.

1. *Broadcasting to diagonals:* Both the row and column broadcast take 2 steps for every pair of rows and columns transferred to the diagonals. Thus, it yields an average of 1 step per row/column. Therefore, the broadcasting overhead is $O(r)$ steps.
2. *Aligning:* The aligning overhead is 1 read-write step per column. $\Rightarrow O(r)$steps.
3. *Multiplication:* This takes $O(r^2/2)$ steps by virtue of the parallelism in computing components of the elements due to using 2r processors.
4. *Routing:* Routing overhead is $O(r)$ steps.
5. *Addition:* This is done in $O(r^2/2 + r)$ steps. There is some speed-up here by the use of 2r processors. By modifying the algorithm to do all additions in the end, a better processor utilization is also possible.

   $O(r(r/2 + 1))$ ˜ $O(r^2/2)$ for $r \gg 1$

   Thus the communications overhead here is $O(3r)$ steps.

Now consider the $MDA(n, 2n, \{0, n\})$.

1. *Broadcasting:* This takes $O(r)$ steps.
2. *Aligning:* Takes $O(r)$ steps as explained above.
3. *Multiplication:* Done in $O(r^2)$ steps. There is no extra speed-up as there are exactly r processors in the system.
4. *Routing:* Routing overhead is again $O(r)$ steps.
5. *Addition:* Done in $O(r^2)$ steps.

Clearly, the algorithm is close to optimal if compared with the known algorithm for $MDA(n, 2n, \{0, n\})$, for which it is known ([9] and [6]) that matrix multiplication takes $O(r^2)$ for multiplication and addition. There is a communications overhead of $O(2r)$ steps where each step comprises a read and a write. This comes from reading the r rows and reading and writing the r columns. Hence, the algorithm given here is close to optimal since every broadcasting, routing and aligning step is essentially a read and write step. For the MDA with $m = 2n - 2$ and access modes given by $q = 0, 1, m/2, m/2 + 1$, a speed-up of 2 is obtained for addition as well as multiplication, as expected for a system with twice the number of processors than the OMP for the same number of memory modules.

2.6 Illustrative examples

The algorithm presented in this paper is illustrated here with some examples. The examples considered are for $n = 3$ and $m = 6$, that is an OMP with 8 processors and 64 memory modules, and an $MDA(4, 6, \{0, 1, 3, 4\})$ with 16 processors and the same number of memory modules. Figures 1 and 2 show the memory accesses under the possible access modes.

Consider broadcasting the first two rows of matrix A to the forward diagonal in the $MDA(4, 6, \{0, 1, 3, 4\})$. Processors labeled 0 through 7 have access to the first row and processors 8 through 15 have access to the second row. It turns out that in the first step of the broadcast under mode $q = 0$, the rows are sent as close to the diagonal as is possible under this mode such that the distance of the elements from the diagonal is at most one vertical step away. This distance can be positive (above the diagonal) or negative (below the diagonal). The first row is sent such that its elements are either 0 or +1 step away from the forward diagonal. For the second row, which is accessed by a different set of processors, this distance should be 0 or -1. Thus, for example, PE_0 and PE_1 are masked off, PEs 2, 4, and 6 move elements down to the diagonal memory modules, and PEs 3, 5, and 7 move the elements such that they are in memory modules directly above the desired diagonal memory modules. PEs 8 through 15 work similarly except that now they must move elements such that they are either in the diagonal memory modules or in memory modules directly below the desired diagonal memory modules. This first step is similar for the broadcast of the lower half rows of the matrix A. The second step of the broadcast is done in mode $q = 4$ where the remaining elements are moved to the desired diagonal memory modules.

Column broadcast is done in a similar fashion except that here the modes are $q = 1$ and $q = 4$ respectively. Each column is sent to both diagonals and the distances here would be positive (left) and negative (right) with respect to the diagonals. Routing is accomplished in exactly the reverse sense of the row broadcast. Addition is done in two steps in modes $q = 3$ and $q = 1$ respectively. Placing of addition results depends on the elements being computed. Thus, all steps of the presented algorithm can be accomplished in the four access modes allowed.

Matrix Multiplication on the $MDA(3, 6, \{0, 3\})$ proceeds along similar lines. The difference here is that only one row or column is moved to the diagonal at a time but this is accomplished in just one step, in mode $q = 0$ for row broadcast and in mode $q = 3$ for column broadcast. The rest of the operations are also done in only one of the two allowed modes. The behavior of the algorithm is very similar to the previous one.

A careful observation of Figures 1 and 2 reveals that each diagonal is accessed by the same set of processors. This fact was used in the algorithm presented. Also, as explained above, the process of broadcasting to the rows comprises steps which take the elements as close to the diagonal as is possible under that mode. For row broadcast, this can be thought of as a vertical distance between the starting position of an element and its destination, i.e. the diagonal memory module. If the vertical distance is d, the broadcast proceeds in steps such that for each element this distance reduces as $d, -(d-1), +(d-2), .., 0$ where + indicates above the diagonal and − indicates below the diagonal. There are $k/2$ such steps as explained before. For column broadcast, the situation is very similar with the difference that distances now are horizontal and hence left or right.

3 Fast Fourier Transform

In many applications of digital signal processing and image processing, it is necessary to compute the Discrete Fourier Transform (DFT) of a discrete input sequence. If the input sequence is $\{x(n)\}$, then its N-point DFT is given by $\{X(p)\}$, where N is the length of the input sequence. The expression for computing the DFT is

$$X(p) = \sum_{n=0}^{n=N-1} x(n) * W_N^{np} \quad p = 0, 1, .., N-1$$

and is usually computed using the Fast Fourier Transform (FFT) algorithm. Either decimation in frequency (DIF) or decimation in time (DIT) can be used. For purposes of illustration, we have chosen a DIF FFT algorithm.

3.1 DIF FFT on MDA architectures

The DIF method [7], involves splitting the original N-point sequence into two $N/2$-point sequences consisting of the first and last halves of the original input. The computation is carried out in $log_2 N$ separate stages. At the k^{th} iteration the $N/2$ butterfly operations are performed between pairs of elements which are $N/2^k$ apart. After the butterfly computation one element in each pair must be post-multiplied by some power of a weighting factor W_N, where $W_N = e^{-j2\pi/N}$. Assuming that the proper weighting factor for element $x(i)$ at stage k is stored at $W(i)$, the necessary computations at iteration k take the form:

$$x(i) \leftarrow x(i) + x(i + \frac{N}{2^k})$$

$$x(i + \frac{N}{2^k}) \leftarrow [x(i) - x(i + \frac{N}{2^k})]\ W(i + \frac{N}{2^k})$$

Consider the problem of mapping the above computation onto a two-dimensional array of 2^m memory modules which can be accessed by 2^n processors in certain modes. This problem was solved for OMP and reported in [6]. For our purposes we need to derive a mapping which can scale as we change the number of processors in the system. With proper initial storage of the powers of W_N, these factors can be easily routed to the correct processors at each stage. For reference, Figure 3 shown an $MDA(3, 4, \{0, 1, 2, 3\})$, in a manner akin to Figures ??, and the proposed storage scheme for the weighting factors is shown in Figure 4 for the case of a 16-point FFT. Initially, the array of memory modules contains the powers of W_N in the order

$0; \frac{0N}{2}; \frac{0N}{4}, \frac{1N}{4}; \frac{0N}{8}, \frac{1N}{8}, .., \frac{3N}{8}; ..; \frac{0N}{N}, \frac{1N}{N}, .., (\frac{N}{2} - 1)\frac{N}{N}$

When the powers of W_N are arranged in this manner, with $W(i)$ stored along with $x(i)$ in row major form in the memory array, routing the weighting factors reduces to copying the factor stored at $W(i)$ to $W(i + \frac{N}{2^k})$ after iteration stage k. During stage k computations are carried out between pairs of memory elements with indices that differ only in the $(m-k)th$ bit. Assuming initial storage of the sequence x and the weighting factors W, the generalized algorithm for the DIF FFT consists of the following steps:

```
for $k = 1$ to m
do
    for all index pairs $i$ and $i + 2^{m-k}$
    do
        $x(i) \leftarrow x(i) + x(i + 2^{m-k})$
        $x(i + 2^{m-k}) \leftarrow [x(i) - x(i + 2^{m-k})]\ W(i + 2^{m-k})$
        $W(i + 2^{m-k}) \leftarrow W(i)$
    loop
loop
```

Figures 5 and 6 show, in an unfolded manner, the iterations needed for and FFT in the $MDA(3, 4, \{0, 1, 2, 3\})$ and the $MDA(2, 4, \{0, 2\})$. This algorithm can be mapped onto any connected $MDA(n, m, Q^*)$. All of the necessary data accesses at each stage k of the FFT can be accomplished in a single mode, since memory elements that are $\frac{N}{2^k}$ apart can always be accessed by a single processor in at least one mode. An omega graph provides the minimal set of modes, with only one mode accomplishing the necessary accesses at each stage. Recall that each mode q in

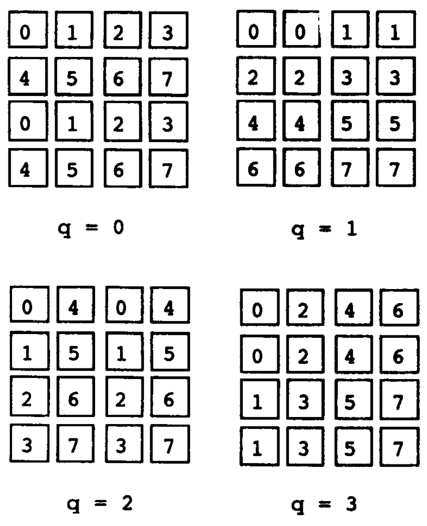


Figure 3: MDA(3,4,{0,1,2,3})

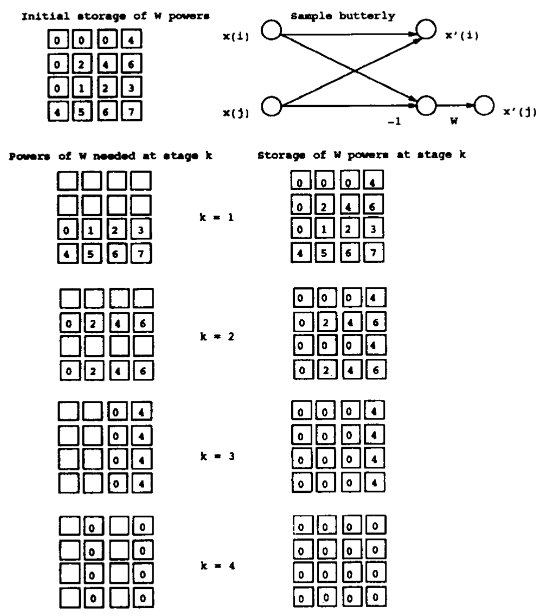
an omega graph allows processor access to modules whose indexes differ in $m-n$ contiguous bits. Thus, each mode must be used to perform $m-n$ consecutive stages in the DIF FFT. The access modes in an omega graph must satisfy $(q_{l+1}-q_l) \bmod m = m-n$. For simplicity, assume $q_0 = 0$, which gives the relation $q_l = l(m-n) \quad l = 0,1,\ldots,\omega-1$, where ω denotes the cardinality of Q^* and satisfies $\omega = \frac{m}{m-n}$ [8]. Note that any omega graph with q_0 in the range 1 to $(m-n-1)$ is isomorphic to the graph with $q_0 = 0$.

In the omega graph, mode $q_0 = 0$ allows access to the $m-n$ most significant bits of the m-bit indices. Thus, mode q_0 must be used for the first $m-n$ stages of the FFT computation. The next $m-n$ bits can be accessed in mode $q_{\omega-1}$, so the next $m-n$ stages take place in this mode. Thus, the modes must be used in the order $q_0, q_{\omega-1}, q_{\omega-2}, \ldots, q_2, q_1$, with each mode used for $m-n$ consecutive stages. The minimal set of modes can provide all of the needed accesses for the computations and routing in the DIF FFT algorithm. Additional modes do not improve the performance of the algorithm; the extra modes simply provide more than one possible mode to use at some of the stages.

3.2 Complexity analysis

Note that $MDA(n,m,Q^*)$ performs the FFT on an N element input, where $N = 2^m$, with p processors, where $p = 2^n$. The minimum required number of access modes is $\lceil \frac{m}{m-n} \rceil$.

- *Total number of butterfly computations* $= \frac{N}{2}\log_2 N = m2^{m-1}$
- *Number of butterfly computations completed in a single step* $= p = 2^n$*, since no processor need ever be idle; thus, number of steps* $= \frac{N\log_2 N}{2p} = m2^{m-n-1}$



Initial storage of W powers
Sample butterly
x(i)
x'(i)
x(j)
x'(j)
-1
W
Powers of W needed at stage k
Storage of W powers at stage k
k = 1
k = 2
k = 3
k = 4

Figure 4: Storage of the powers of W_N at each stage k

The case $m-n=1$ *generates the hypercube. In this case, the number of steps reduces to simply* $\log_2 N$ *because all* $N/2$ *butterflies at each stage can be computed in parallel by the* $N/2$ *processors. The OMP architecture, corresponding to* $m = 2n$, *requires* $\frac{\sqrt{N}\log_2 N}{2}$ *steps.*

- *Total number of multiplies* $= \frac{2N\log_2 N}{p} = 2m2^{m-n}$*, with four real multiplies (one complex multiply) for each butterfly*
- *Total number of additions* $= \frac{3N\log_2 N}{p} = 3m2^{m-n}$*, with six real additions (three complex additions) for each butterfly*

  To decrease the number of complex multiplies and additions, do not postmultiply in the final stage. The last iteration involves postmultiplying by $W_N^0 = 1$.
- *Communication overhead* $= O(\frac{N\log_2 N}{2p}) = O(m2^{m-n-1})$*, since a W factor must be copied between each butterfly pair in each stage.*

From the analysis above we see that this fully-scalable algorithm provides optimal speed-up of $O(p)$. In this case, the processor utilization is also very good since all processors work in parallel in all the stages of the FFT computation. The number of stages under each mode, the number and type of access modes and the computation time can be specified in terms of the parameters m and n.

4 Conclusions

From the preceding discussion, it is evident that generalized and scalable algorithms exist for orthogonal shared memory multiprocessing systems. These algorithms map

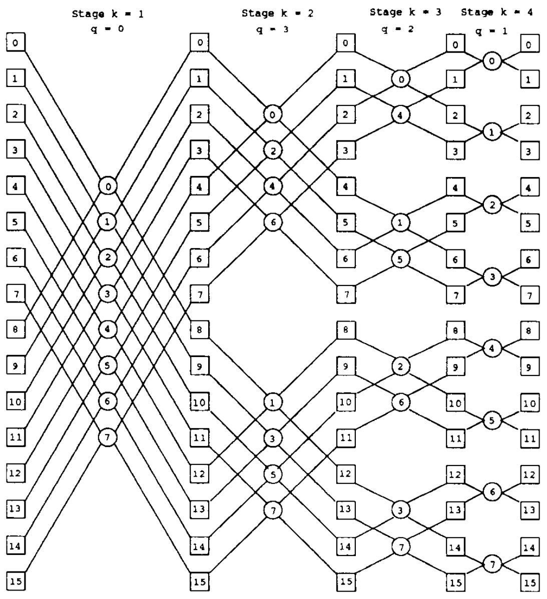


Figure 5: 16-point DIF FFT on MDA(3,4,{0,1,2,3})

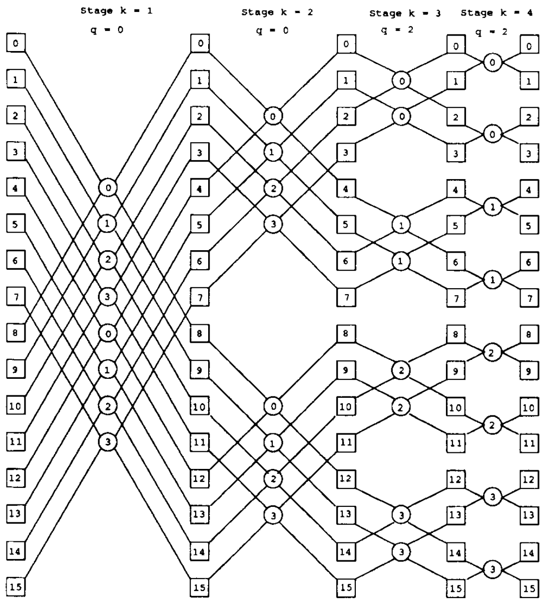


Figure 6: 16-point DIF FFT on MDA(2,4,{0,2})

well on to the OMP and the hypercube as well as other MDA architectures. By using a graph theoretical framework, we were able to define and analyze two important problems, namely Matrix Multiplication and the FFT. For latter, we saw that the algorithm is scalable and optimal: it gives a speed-up of $O(p)$ for p processors in the system. Also, the algorithm can be analyzed completely by using the connectivity provided in omega graphs. Omega graphs use the minimum number of elements (modes) in the set $Q*$ to guarantee full coverage and the access modes are disjoint. The operation of FFT can be accomplished using this minimal set of modes. However, for matrix multiplication, a different approach was needed to make the algorithm scalable. To ensure scalability, we had to go beyond omega graphs and look for enhanced connectivity in the interconnection network. This gave us an empirical relation $m = 2n - (k-2)$, $k = 2, 4, 6, .., m$, which was very useful in analyzing the given algorithm. We also saw that the above relation specified exactly the kind of connectivity needed in the corresponding orthogonal graph to implement the algorithm. The architecture is completely specified by the value of k. k access modes are required for implementing the proposed algorithm on a particular MDA architecture. Once the value of k is known, the communications overhead also becomes known. Although scalable, the algorithm for matrix multiplication is not optimal. It is optimal for the OMP but as the architecture is scaled further, the processor utilization suffers. This is probably an indication of an existing trade-off between full scalability of an algorithm and its optimality. In the OMP, for example, the known algorithm for matrix multiplication is optimal but it is not scalable. This trade-off between scalability and optimality has several important implications in the design of highly parallel computing structures and in the development of efficient architecture-independent algorithms. We believe that the theory of orthogonal graphs provides a powerful tool which can be used to define, analyze and characterize a large class of orthogonal shared-memory multiprocessing systems.

Acknowledgement

The contributions of Atul Ahuja to the development, analysis and discussion of the matrix multiplication algorithm are appreciated.

References

[1] H. M. Alnuweiri and V. K. P. Kumar, *A Reduced Mesh of Trees Organization for Efficient Solution to Graph Problems,* 22nd Annual Conference on Information Science and Systems, March 1988.

[2] K. E. Batcher, *The Multidimensional Access Memory in STARAN,* IEEE Transactions on Computers, Vol. C-26, No. 2, February 1977, pp. 172-177.

[3] L. N. Bhuyan and D. P. Agrawal, *Generalized Hypercube and Hyperbus Structures for a Computer Network,* IEEE Transactions on Computers, Vol. C-33, No. 4, April 1984, pp. 323-333.

[4] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing,* Reading, McGraw Hill, 1984.

[5] K. Hwang and D. Kim, *Generalization of Orthogonal Multiprocessor for Massively Parallel Computation,* Proceedings of Frontiers 88, 2nd Symposium on the Frontiers of Massively Parallel Computation.

[6] K. Hwang, P. S. Tseng and D. Kim, *An Orthogonal Multiprocessor for Large-Grain Scientific Computations,* IEEE Transactions on Computers, Vol. C-38, No. 1, January 1989, pp. 47-61.

[7] N. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Second Edition, Reading, Springer-Verlag, 1982.

[8] I. D. Scherson, *A Theory for the Description and Analysis of a Class of Interconnection Networks*, Princeton University, Technical Report, CE-S89-002.

[9] I. D. Scherson and Y. Ma, *Analysis and Applications of the Orthogonal Access Multiprocessor*, Journal of Parallel and Distributed Computing, Vol.7, No.2, October 1989.

[10] I. D. Scherson, *Definition and Analysis of a Class of Spanning Bus Orthogonal Multiprocessing Systems*, Proceedings of the ACM 1990 Computer Science Conference, February 1990, Washington D.C., pp. 194-200.

[11] I. D. Scherson, *Orthogonal Graphs and the Analysis and Construction of a Class of Multistage Interconnection Networks,* Proceedings of the 1990 International Conference on Parallel Processing, August 1990.

[12] I. D. Scherson and P. F. Corbett, *Communications Overhead and the Expected Speed-up of Multidimensional Mesh-Connected Parallel Processors,* The Journal of Parallel and Distributed Computing, in press.