DESIGNING ARRAYS FOR THE FADDEEV ALGORITHM

Jaime H. Moreno
Tomas Lang

March 1988
CSD-880013

Designing Arrays for the Faddeev Algorithm

Jaime H. Moreno, Tomás Lang *
Computer Science Department
University of California, Los Angeles
3680 Boelter Hall
Los Angeles, Calif. 90024

*J. Moreno has been supported by an IBM Computer Sciences Fellowship. This research has also been supported in part by the Office of Naval Research, Contract N00014-83-K-0493 "Specifications and Design Methodologies for High-Speed Fault-Tolerant Algorithms and Structures for VLSI"

Abstract

We present the derivation of arrays for computing the Faddeev algorithm, using a graph–based methodology. We consider implementations for fixed–size problems as well as for partitioned mode. Arrays are obtained by performing transformations on the fully–parallel dependence graph of the algorithm. We derive linear and two–dimensional structures and evaluate the resulting arrays in terms of number of PEs, throughput, I/O bandwidth, utilization of PEs and overhead due to partitioning. We also compare our implementations with other schemes previously proposed.

For fixed–size problems, we show that throughput reaches $2/(3n^2 - n + 2)$ in a linear array, while throughput n^{-1} or $(2n)^{-1}$ can be obtained with two–dimensional arrays. The utilization of such arrays tends to 7/9 for large matrices. In the case of partitioned problems, we propose a two–dimensional scheme that is more efficient and has less overhead than other arrays previously proposed. We show that throughput of both linear and square arrays tends to $(3m)/(7n^3)$, where m is the number of cells, and utilization tends to optimal. Moreover, we show that in partitioned implementations with the same number of cells a linear array performs better, is easier to implement and is better suited for fault-tolerant capabilities than a two–dimensional one.

1 Introduction

The implementation of matrix algorithms as collections of regularly connected processing elements (arrays of PEs) has been extensively studied lately [1]–[7]. A problem frequently found in designing such arrays is that of providing capabilities to compute several algorithms in the same structure, so that an array can be used for different computations. One approach towards solving this problem consists of building into the array the capability to perform a class of matrix algorithms and controlling execution of any of the possible algorithms through programmable features in the array. Processing elements have been designed with such goal in mind [8,9] and systems have been built offering a variety of matrix computational capabilities [10]–[13].

A different approach to offer flexibility in an array of PEs consists of using a general–purpose algorithm within a class of problems. Such is the case of the Faddeev algorithm [14], which has the capability of performing a variety of matrix computations without the need for programmable features. Some overhead or cost is involved of course, which in this case consists of performing additional computations. Several arrays to compute the Faddeev algorithm have been discussed in the literature [15]– [18]. Nash and Hansen [15] have proposed a trapezoidal array for fixed–size problems and an implementation of their scheme has been presented in [16]. The same structure is used in [17] to compute the Faddeev algorithm for matrices larger than the size of an existing array (i.e., partitioning the problem so that it fits into an array). The Faddeev algorithm is also implemented in [18] for both fixed size and variable size problems, and in [19] for partitioned implementation in a two–dimensional array of transputers.

The arrays to compute the Faddeev algorithm mentioned above have not been derived through the application of a design methodology (this is also the case with many schemes for other matrix computations), though methods to derive the structure and interconnection of arrays of PEs for a given algorithm have been proposed [20]. The existing methods are transformational approaches, where a high–level specification of a problem is transformed into a form better suited for implementation. Such mechanisms can be useful to accomplish certain design tasks, but they either restrict the form of the algorithm (i.e., a recurrence equation), are unable to incorporate parameters of interest for the implementation, or are too complex to use. Thus, it is not surprising that they have not been used in the derivation of arrays for the Faddeev algorithm.

In this paper, we present the application to the Faddeev algorithm of the design methodology proposed in [21]–[23]. This is a transformational method, which uses a fully–parallel dependence graph as the description of the algorithm. The graphical nature of this methodology makes it easier to use than approaches based on mathematical expressions and the dependence graph doesn't restrict the form of the algorithm. We use such methodology to derive linear and two–dimensional structures for the Faddeev algorithm, which take advantage of different properties in the computation. We discuss the design of arrays for fixed–size problems, as well as the computation of the algorithm for matrices larger than the size of an array (i.e., partitioning the algorithm). We compare the structures obtained in terms of number of PEs, throughput, input bandwidth, utilization of PEs and overhead due to partitioning.

We show that for matrices of size n by n it is possible to achieve throughput $2/(3n^2 - n + 2)$ with $O(n)$ cells organized as linear structures and throughput $[n]^{-1}$ or $[2n]^{-1}$ in two–dimensional arrays with $O(n^2)$ cells. The utilization of such arrays tends to 7/9. One of the two–dimensional schemes derived here corresponds to the one proposed by Nash and Hansen in [15], though their

paper didn't include an evaluation of such array in terms of throughput and utilization as it is possible with our graph–based methodology.

In partitioned mode, we show that for very large matrices the throughput of linear and two–dimensional arrays tends to $(3m)/(7n^3)$ (where m is the total number of PEs) and utilization tends to 1. The two–dimensional partitioned scheme devised here is more efficient than the one proposed in [18]. In addition, it doesn't need the complex loading and un–loading of data required in [17]. Moreover, we show that a linear array is simpler, has slightly better throughput and utilization with the same number of units than a square array, and exhibits better characteristics for fault–tolerant implementations than a two–dimensional structure.

2 The Modified Faddeev Algorithm

The matrix version of the Faddeev algorithm evaluates the expression $CX + D$ subject to the condition $AX = B$, where A, B, C, D are given matrices, X is a column vector, and A is of full rank. The algorithm can be expressed by representing the data as the extended matrix

$$\begin{array}{c|c} A & B \\ \hline -C & D \end{array}$$

and performing linear combinations on this extended matrix with the objective of transforming matrix C into a matrix of zeroes. If we represent such linear combinations as W, the operations performed are $(-C + WA)$ and $(D + WB)$. The annulment of C requires that $W = CA^{-1}$, so that $D + WB = D + CA^{-1}B$. Since $X = A^{-1}B$, the final result $D + WB = D + CX$ replaces the values of matrix D in the expression above [15].

Several matrix operations are possible by selecting specific entries for matrices A, B, C and D. Figure 1 depicts such alternatives, which include matrix multiplication/addition, matrix inversion and solution of linear systems of equations. Therefore, the Faddeev algorithm allows a degree of "programmability" by selecting the values of the input data. The cost of such flexibility consists of the additional operations that need to be performed, because the algorithm operates on four matrices.

In addition to its capability to perform different matrix operations, the Faddeev algorithm has other advantages:

- The algorithm doesn't need to compute the elements of W, since it only needs to annul the elements of C. Such annulment can be done by ordinary Gaussian elimination.
- When solving linear systems of equations, the algorithm doesn't need the back–substitution step usually found in triangularization methods. Instead, results are obtained directly at the end of the annulment of C.

Since Gaussian elimination doesn't guarantee numerical stability and fails for zero pivot elements, Nash and Hansen have proposed a modification to the original Fadeev algorithm [15]. Such modification consists of "adding an orthogonal factorization capability for added numerical stability and to allow the coefficient matrix to be non–square for over– and under–determined systems of

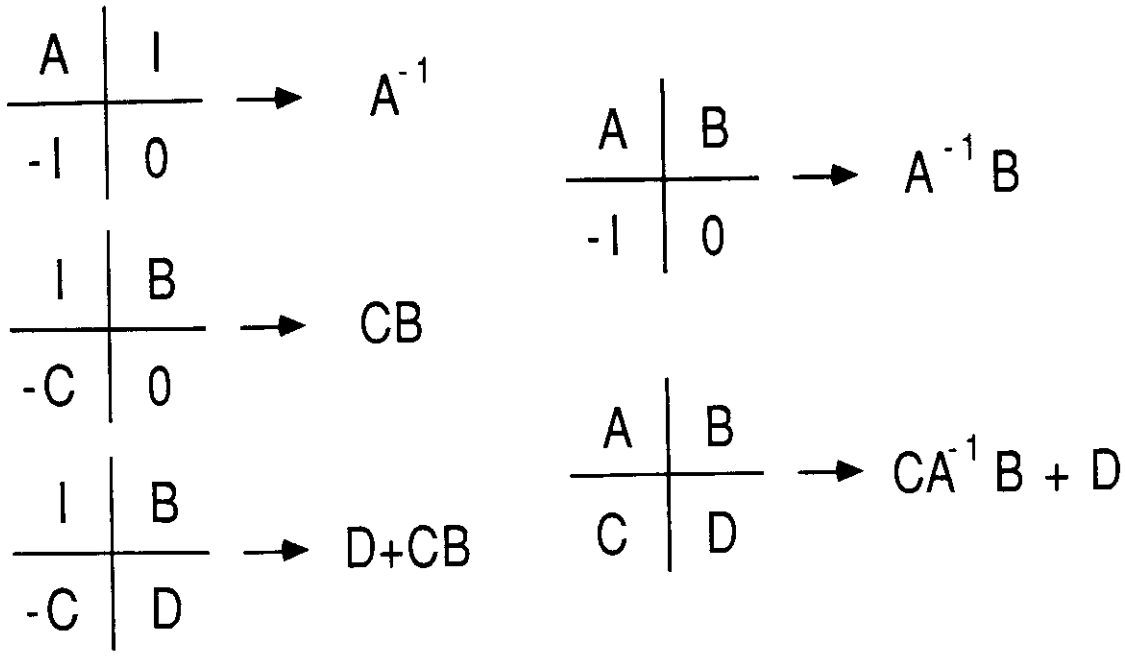


Figure 1: Examples of matrix operations available with the Faddeev algorithm

equations". Nash and Hansen's scheme uses Givens rotations to annul matrix C. The utilization of Givens rotations requires to divide the process of annulling matrix C into the following two-step procedure [15]:

- Triangularization of matrix A through Givens rotations and application of such rotations to matrix B.
- Gaussian elimination of the elements of C using the diagonal elements of the rotated matrix A (i.e., matrix Q) as pivots and application of the same transformations to D.

Figure 2 shows the dependence graph of the modified Faddeev algorithm for 4 by 4 matrices, after replacing data broadcasting by data pipelining [21,22]. Delay nodes have been added to enhance communications regularity between nodes of the graph and to obtain nodes with at most one external input. Operation nodes correspond to the computation of multiply/add, division, rotation angle and rotation. For simplicity, we assume that all these nodes have the same computation time. The validity of such an assumption is highly implementation-dependent, as recent studies about the design of special-purpose cells suggest [24,25,26].

We can distinguish four sections in Figure 2, namely those used to operate on the four different matrices. In the top-left section, diagonal elements of matrix A are used to compute rotation angles. Such angles are broadcasted horizontally to the remaining elements of A on the same row and to the elements of B also on the same row. All these elements are rotated according to such angles. Elements of the resulting triangular matrix Q and the rotated matrix B' flow towards the lower sections of the graph. In the lower-left section, diagonal elements of Q are used as pivots to perform Gaussian elimination on matrix C. Pivots are broadcasted horizontally and used together with elements of B' to perform the same transformation on matrix D.

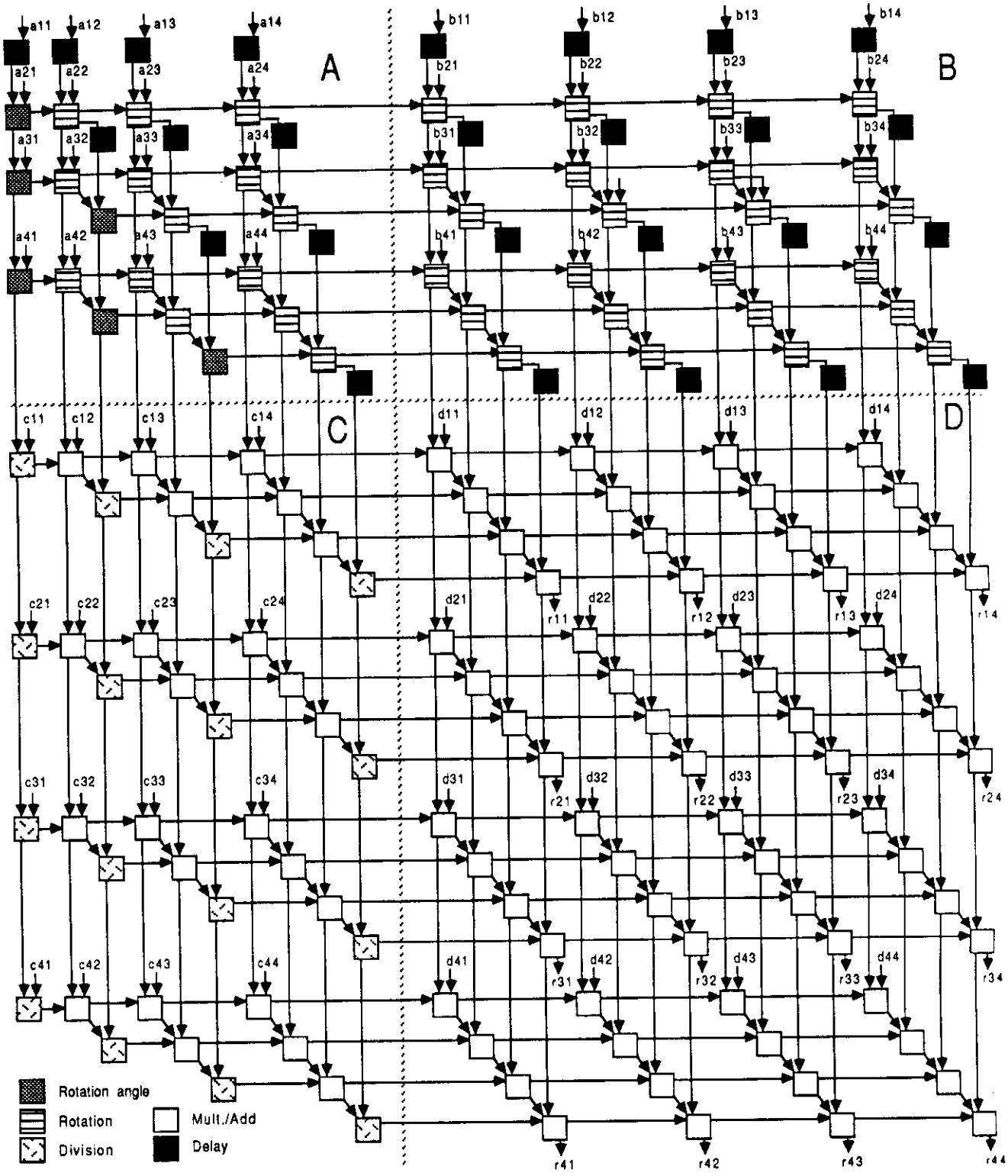


Figure 2: Fully-parallel dependence graph for the Faddeev algorithm with no broadcasting

In the next sections, we use the dependence graph described above to discuss the design of arrays for the Faddeev algorithm.

3 Arrays for the Faddeev algorithm with fixed–size matrices

In this section, we study the design of arrays of processing elements (PEs) for the Faddeev algorithm with fixed–size matrices, using the dependence graph in Figure 2 as description of the algorithm. A pipelined implementation of such graph [21,22] leads to a structure with $O(n^3)$ nodes, $O(n^2)$ input bandwidth and complex interconnection, undesirable features for an implementation. We use the methodology proposed in [21]–[23] to transform the graph and incorporate into it implementation restrictions.

3.1 Linear arrays for the Faddeev algorithm with fixed–size matrices

Let's assume that we want to obtain an implementation for the Faddeev algorithm with $O(n)$ cells. Since the graph in Figure 2 has $O(n^3)$ nodes, we have to transform the graph by grouping nodes such that the transformed graph has $O(n)$ nodes. We have proposed a procedure to reduce the number of nodes in dependence graphs [23], which collapses sub–paths of a graph into single nodes. However, such procedure is complicated in this case because it is not possible to group all nodes with just $O(n)$ subpaths of the same length, neither to find $O(n)$ groups with the same number of nodes without compromising the communications structure of the graph. These facts lead to groupings with different number of nodes per group and consequently to implementations with different computation time per cell and non–optimal utilization. Thus, we must attempt to group sub–paths in such a way as to minimize the difference in the number of nodes per group.

From the graph we can recognize $O(n^2)$ vertical (or horizontal) sub–paths suitable for collapsing. In addition, there are $O(n^2)$ diagonal sub–paths*. We can't group diagonal sub–paths into single nodes because this would lead to $O(n^2)$ nodes and we are interested in $O(n)$ cells. We have to look for a grouping which collapses $O(n)$ diagonal sub–paths into a single group. We also want to minimize the difference in the number of nodes per group without unduly increasing the communication complexity. Therefore, we choose to group vertical (or horizontal) sub–paths in such a way as to include each diagonal sub–path in a single group, leading to $2n$ groups.

The grouping discussed above is shown in Figure 3 for the case of grouping vertical sub–paths. The graph resulting from such grouping requires that the leftmost node be capable of performing division and computing rotation angles, the next $n-1$ nodes must perform those two operations plus rotation and multiplication/addition, while the remaining n nodes perform only rotation and multiplication/addition.

Throughput achievable with the grouping shown in Figure 3 can be obtained directly from the graph. Such throughput is determined by the largest group (i.e., the group with the largest number of primitive nodes), because such group has the longest computation time. In this case, it corresponds to the rightmost group. A cell implementing this group must perform $\frac{1}{2}n(n-1)$

*Diagonal sub–paths actually correspond to a third dimension of the graph. For simplicity, we have drawn the graph in a two–dimensional plane and projected the third dimension onto such diagonal sub–paths

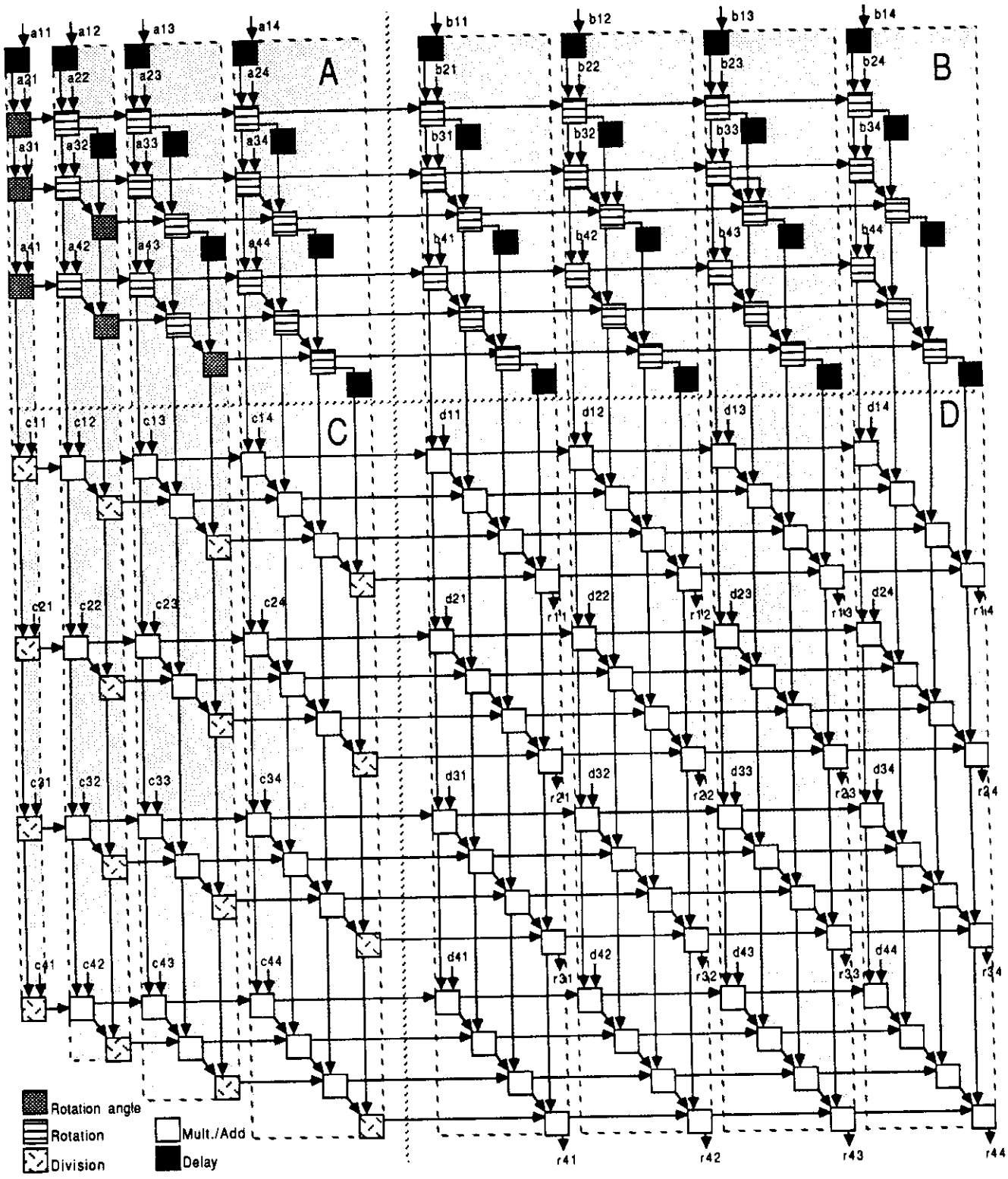


Figure 3: Grouping vertical sub-paths in Faddeev dependence graph

rotations, n^2 multiply/add operations, and n delays, leading to throughput

$$T_{\text{linear}} = \left[\frac{1}{2}n(n-1) + n^2 + n\right]^{-1} = \frac{2}{3n^2+n} \quad [\text{ops}]^{-1}$$

Total computation time for one instance of the algorithm can be expressed as the time to compute the last (i.e., the rightmost) group, plus the delay before starting the computation of such group. Thus, total computation time is

$$t_{\text{linear}} = \frac{1}{2}n(n-1) + n^2 + 3n - 1 = \frac{3}{2}n^2 + \frac{5}{2}n - 1 \approx \frac{n}{2}(3n+5) \quad [\text{ops}]$$

Operations within each group in Figure 3 can be executed in one of the following two orders:

- Schedule nodes of each group by diagonal sub–paths, that is, execute all nodes in a diagonal sub–path before scheduling another diagonal sub–path. In such a case, delay elements have to be added to vertical sub–paths in a group in order to synchronize the arrival of data to nodes. Thus, $O(n)$ delay elements are required per group. Since delay nodes were added to enhance communications requirements of the graph and with this scheduling all data is local to cells, delay nodes shown in Figure 2 don't need to be executed (excepting those delays at the top of the graph). Consequently, throughput can be increased to $2/(3n^2 - n + 2)$. Data arrival to nodes of this graph is interspersed with null values. The transformed graph for such scheme, shown in Figure 4, can be directly mapped onto a linear array.
- Schedule nodes of each group by vertical sub–paths, that is, execute all nodes in a vertical sub–path before scheduling another vertical sub–path. In such a case, intermediate results used within a group need to be fed back into the node, as shown in Figure 5. Input data arrives to nodes of this graph at successive intervals. The graph can be mapped directly onto a linear array.

Cells in the arrays discussed above perform different computations, as inferred from the primitive nodes composing the groups leading to such cells. This is a consequence of the grouping performed, because there were no restrictions on the type of nodes composing a group. An alternative approach considers searching for groups of similar nodes, leading to structures with specialized cells. Since cells are specialized, they might be simpler than cells in arrays with more general PEs. For the linear arrays discussed above, it is possible to obtain specialized cells at the cost of data movement. If we schedule the computation of nodes in each group by vertical sub–paths, the feedback of data into a node described above can be modified so that data is fed back not to the same node but to the node to its left (i.e., intermediate data is shifted to the left). This is equivalent to redrawing the graph in Figure 2 in such a way that all vertical sub–paths with nodes performing division/rotation angle appear at the left of the graph and are grouped together. With this approach, all computations of rotation angles and pivots are performed in the leftmost group. The remaining groups need to perform only rotations and multiply/add operations in Gaussian elimination. The resulting graph, shown in Figure 6, can be directly mapped onto a linear array.

Linear arrays with higher throughput than what is achievable from the structures described above can be obtained by grouping fewer nodes per group. Since the dependence graph is composed

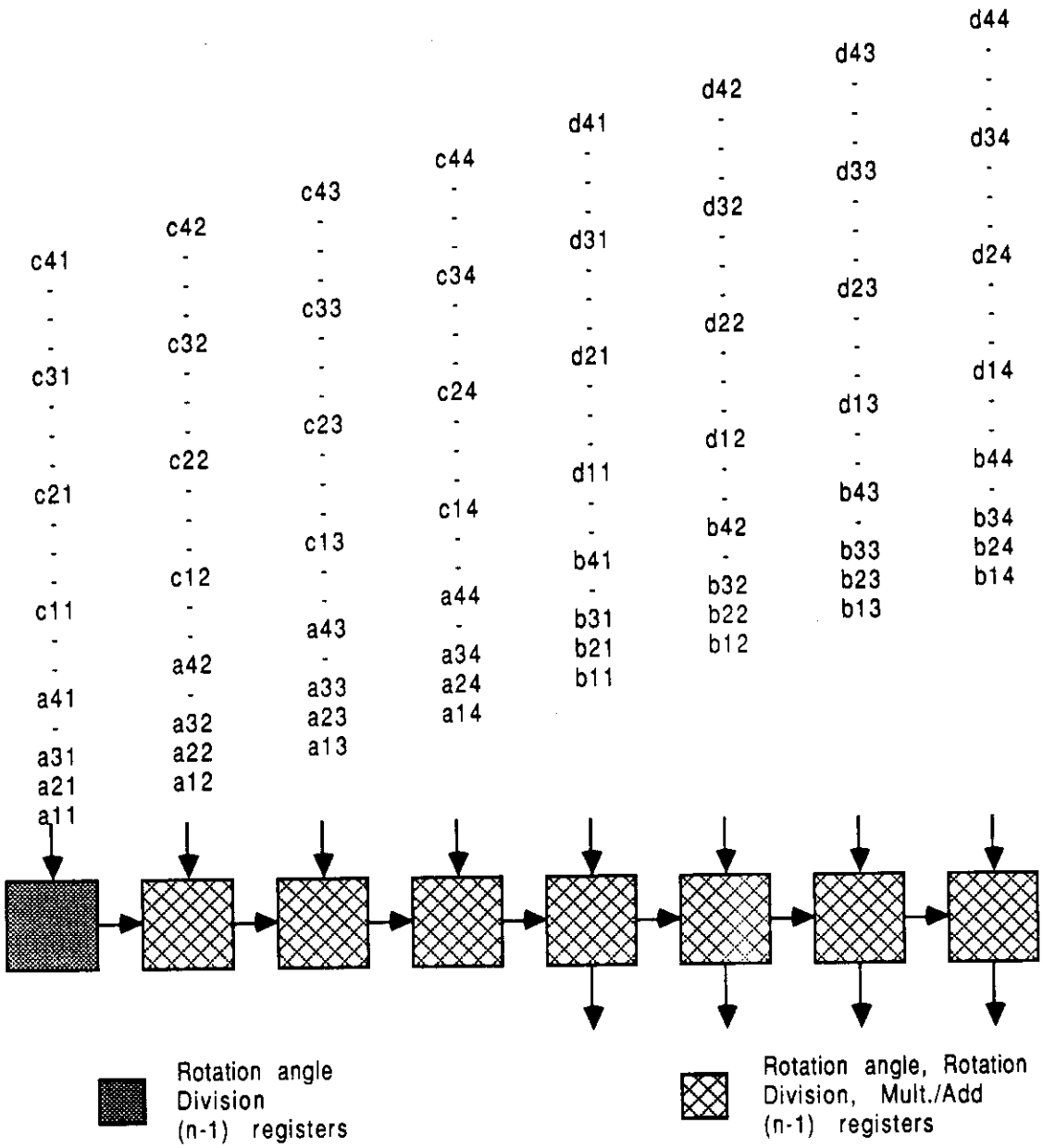


Figure 4: Scheduling nodes by diagonal sub-paths

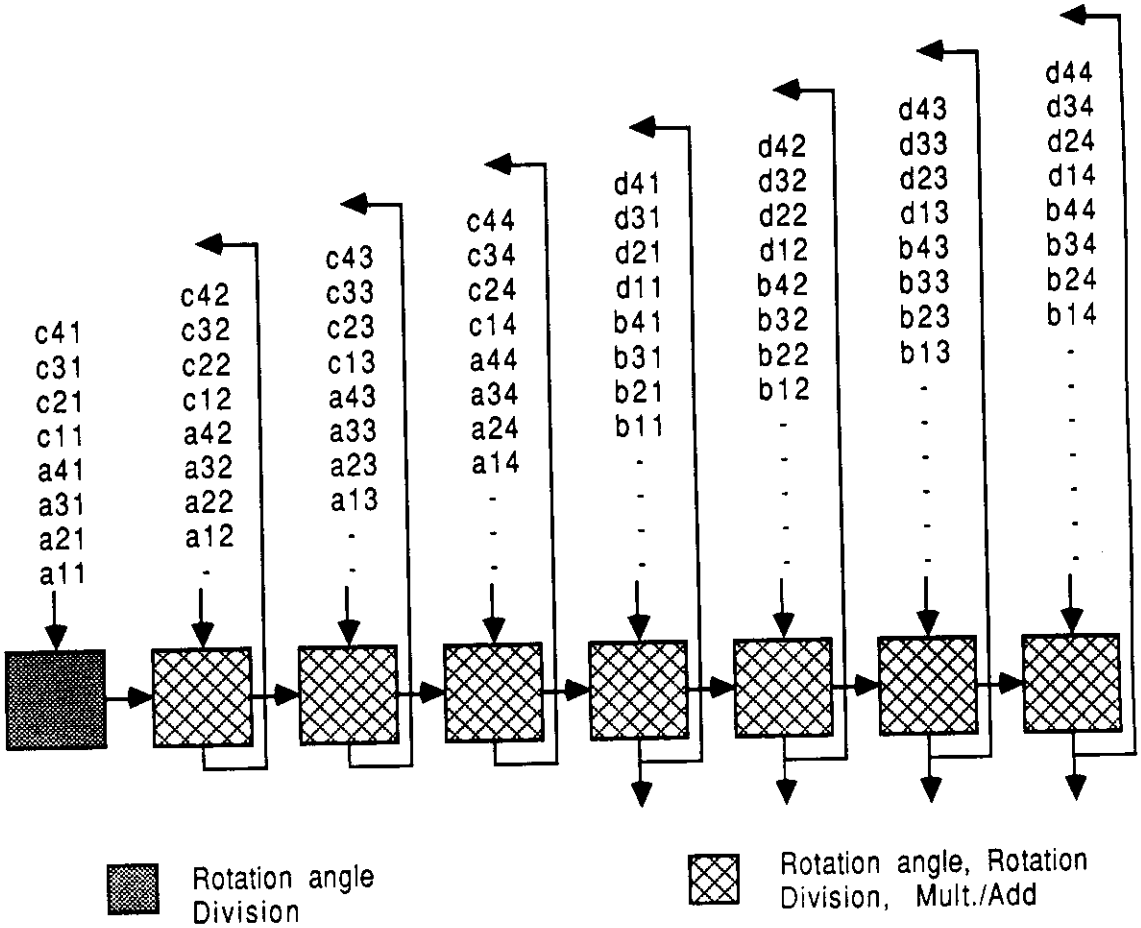


Figure 5: Scheduling nodes by vertical sub-paths

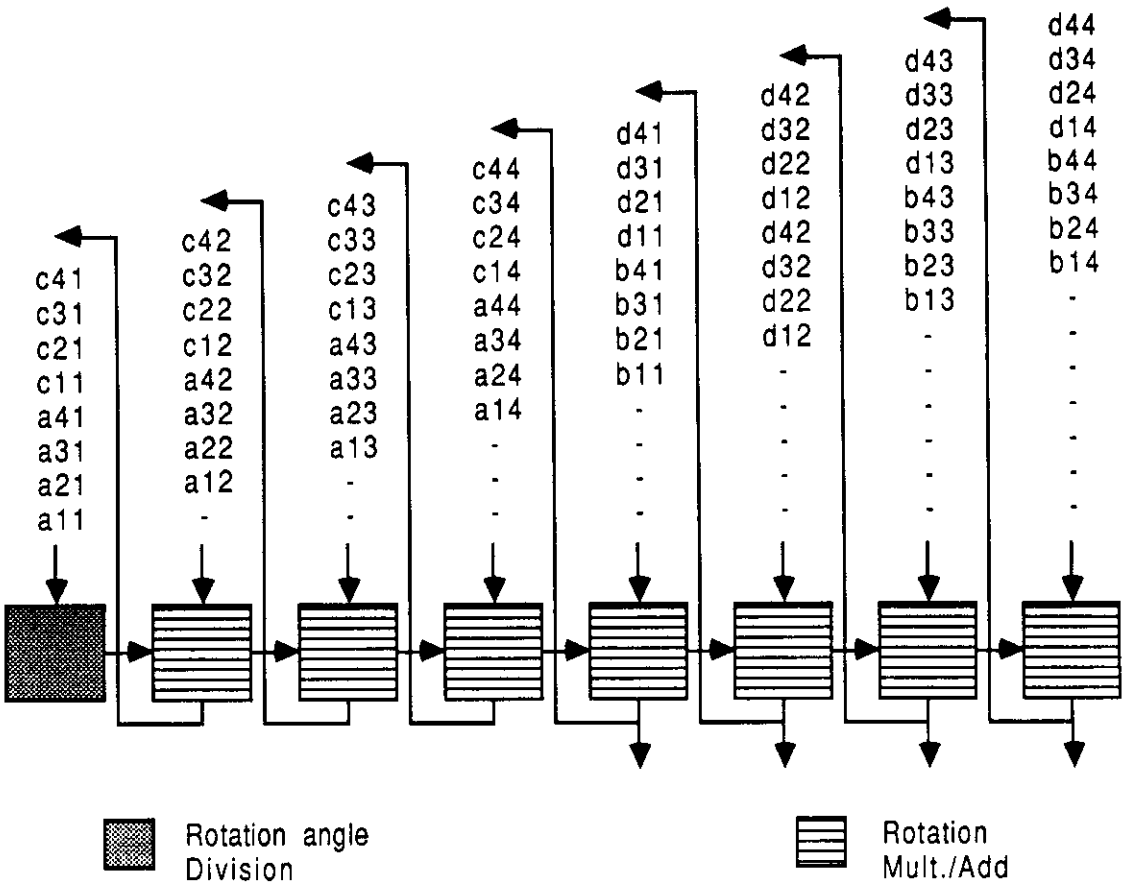


Figure 6: Linear graph with specialized nodes and shifting of data

of different sections, one alternative consists of separating nodes in the two topmost sections of the graph from the bottom ones and group them independently. In such a case, we obtain a linear graph with two nodes in each position. These nodes are simpler than in the previous cases, because topmost nodes perform only triangularization while bottom nodes only perform Gaussian elimination. All approaches discussed above for scheduling the nodes within a group are also suitable for this case. Figure 7 shows the structure of a bi–linear array with shifting of intermediate results, where each cell needs to perform only one type of computation. Throughput of this array is n^{-2}, almost twice that of previous examples, since the largest group has n^2 nodes. Total computation time for one instance of the algorithm is still $\frac{3}{2}n(n+1)$[ops], because the critical path in the graph has not been modified with respect to the previous examples.

3.2 Two–dimensional arrays for the Faddeev algorithm with fixed–size matrices

In this section, we use the dependence graph in Figure 2 to illustrate the design of two–dimensional arrays for the Faddeev algorithm with fixed–size matrices. Since the graph has $O(n^3)$ nodes and we want to obtain $O(n^2)$ cells, we need to perform groupings of $O(n)$ nodes per group. Again, we are concerned with the utilization of the resulting arrays so that we attempt to obtain groups with the same number of nodes. As in the case of linear arrays, the grouping process is performed following the procedure proposed in [23]. We use the extended version of such procedure, which selects groups according to the length of sub–paths in the graph.

There are basically three alternative approaches towards performing grouping of nodes in Figure 2: group vertical, horizontal or diagonal sub–paths. There are several sub–paths of the same (maximum) length in each of these cases, so that we can expect to obtain good utilization of the resulting arrays.

Grouping diagonal sub–paths of the graph: square array

We first perform grouping of diagonal sub–paths of the graph, as shown in Figure 8. Such grouping leads to the transformed graph shown in Figure 9, which can be directly mapped onto a square array. In such an array, the lower rightmost $n(n+1)$ cells are fully utilized while remaining cells have lower utilization. Cells perform different operations, as inferred from the figure. Results are obtained from the lower–rightmost n^2 cells. The transformed graph requires data input in every node, leading to $O(n^2)$ input pads. However, those pads will be under–utilized because only one data element is transferred through each input for each instance of the algorithm. The input bandwidth problem can be solved by decoupling data transfer from computation, as proposed for such case in [21,22].

Since the largest group in the transformed graph is composed of n nodes, this scheme has throughput

$$T_{\text{square}} = \frac{1}{n} \quad [\text{ops}]^{-1}$$

In a similar manner to the case of the linear array in Figure 6, we can simplify nodes in the square graph by moving data towards the upper–left corner. In such a case, only nodes in the

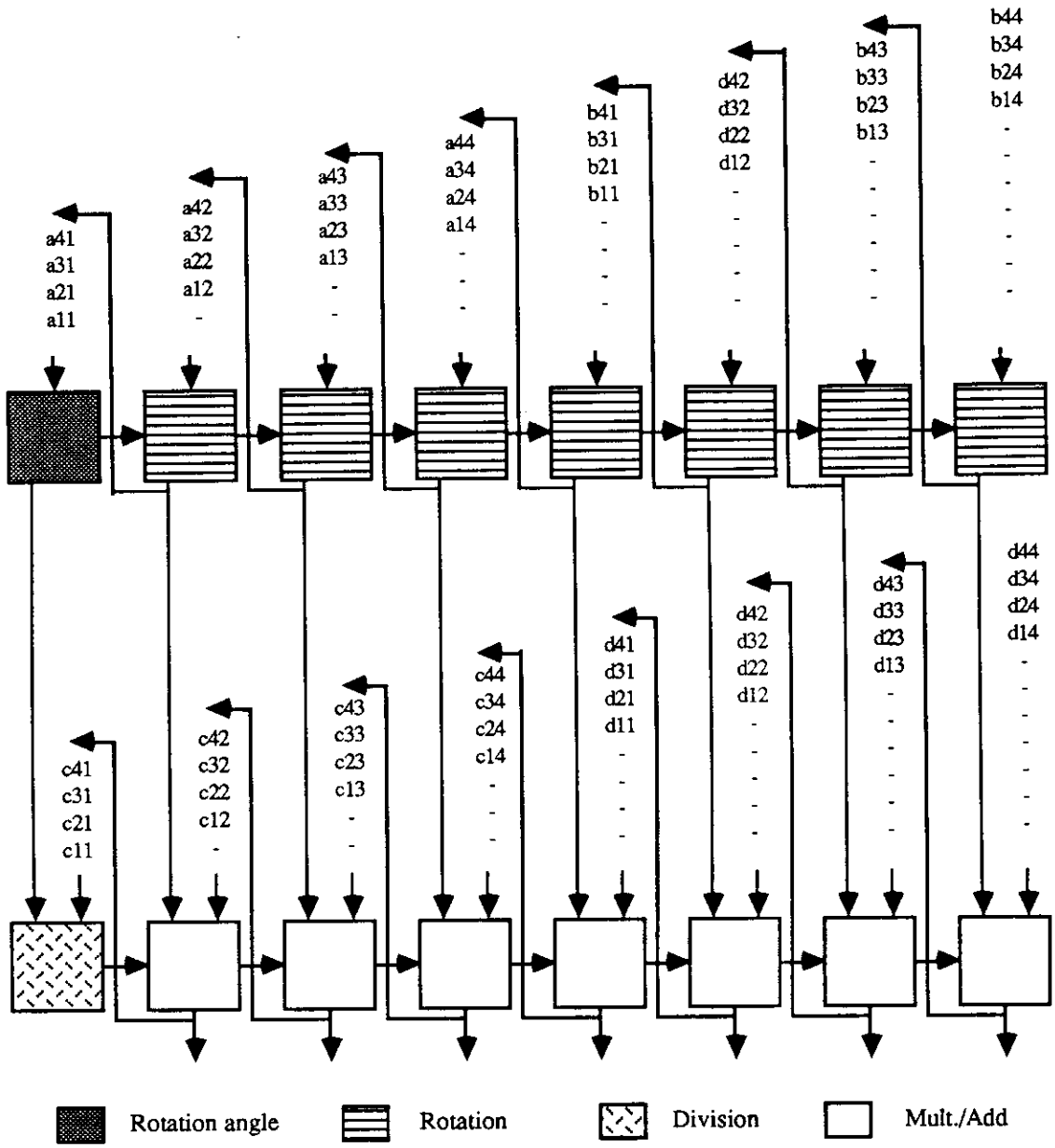


Figure 7: Bi–linear array with specialized cells and shifting of data

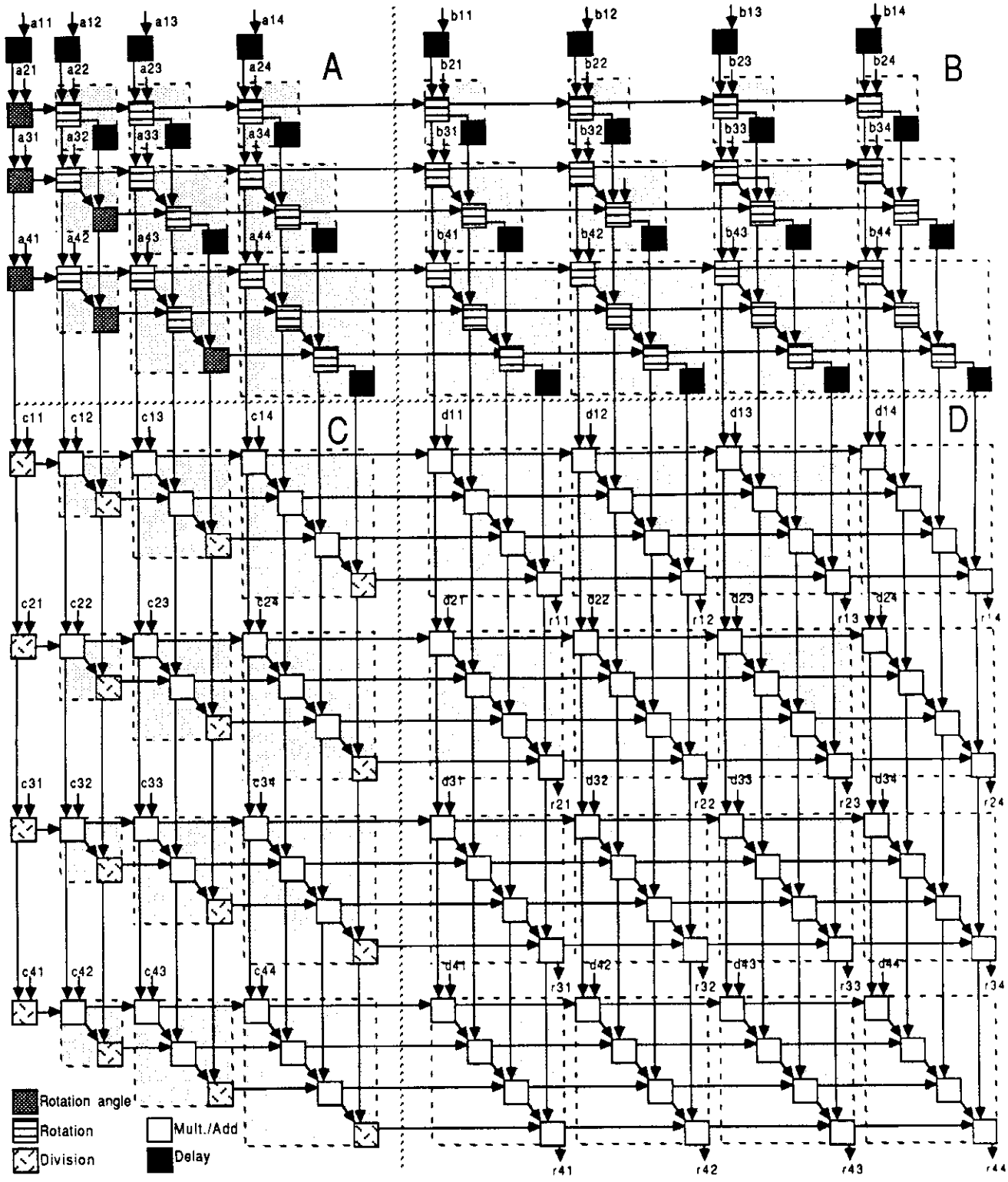


Figure 8: Grouping diagonal sub-paths of the dependence graph

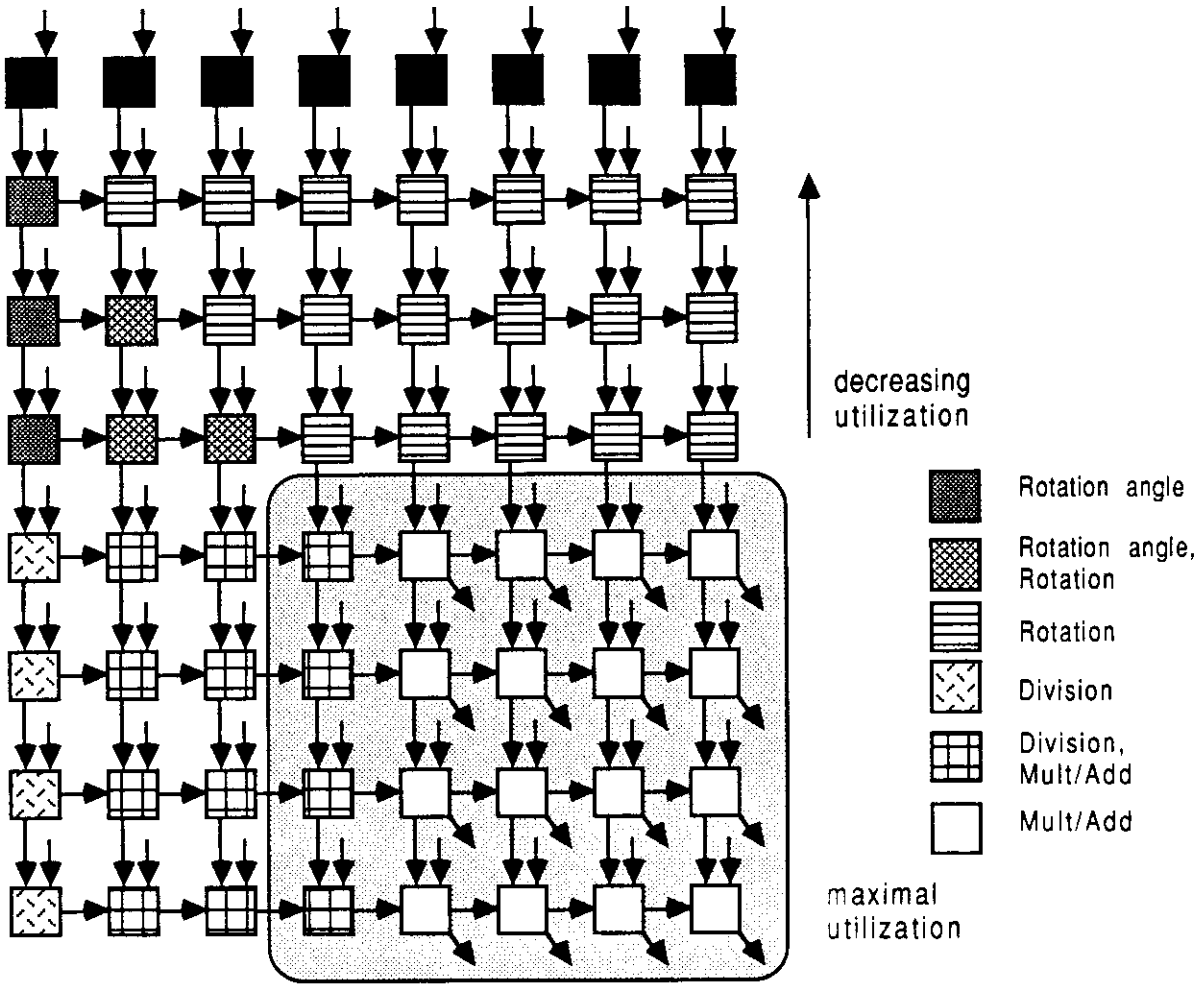


Figure 9: Square graph for the Faddeev algorithm

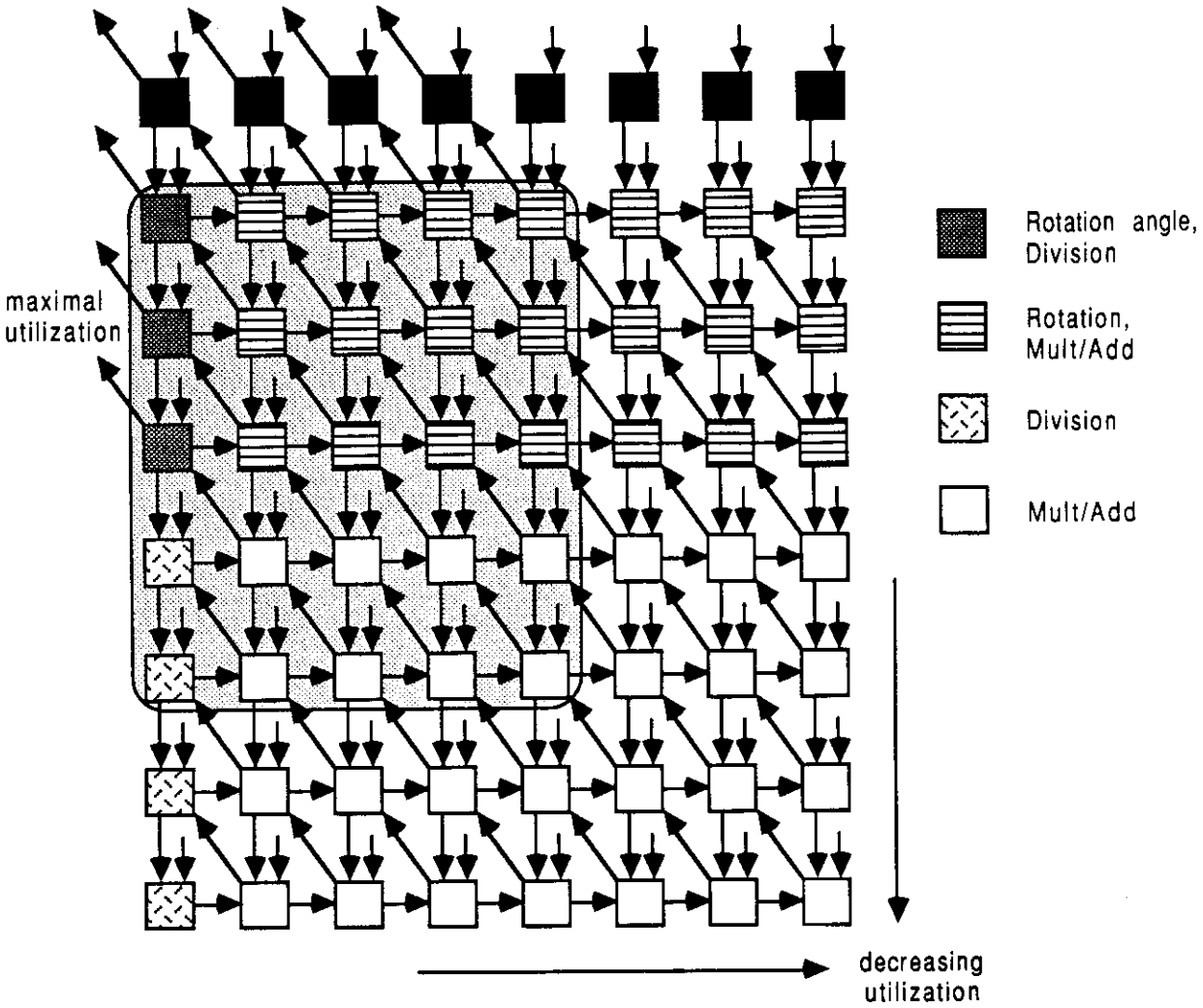


Figure 10: Specialized cells in the square graph

leftmost vertical sub–path of the graph need to perform complex operations such as division and computation of rotation angles. This scheme is shown in Figure 10. Intermediate results flow through the graph and final results can be made available through nodes at the upper–left corner. Consequently, the region of maximal utilization also moves towards the upper–left section of the graph.

Grouping vertical or horizontal sub–paths of the graph: trapezoidal arrays

The result from grouping vertical sub–paths of the graph is shown in Figure 11. In this case, we obtain a graph which can be mapped onto a trapezoidal array. This scheme corresponds to the structure proposed in [15]. Such array has $O(n)$ input bandwidth. Results flow downward through the rightmost n^2 cells of the array. Cells are specialized because only the leftmost ones perform division and computation of rotation angles, while remaining cells perform rotations and multiplication/addition. Utilization of this scheme is not maximal, because the number of operations

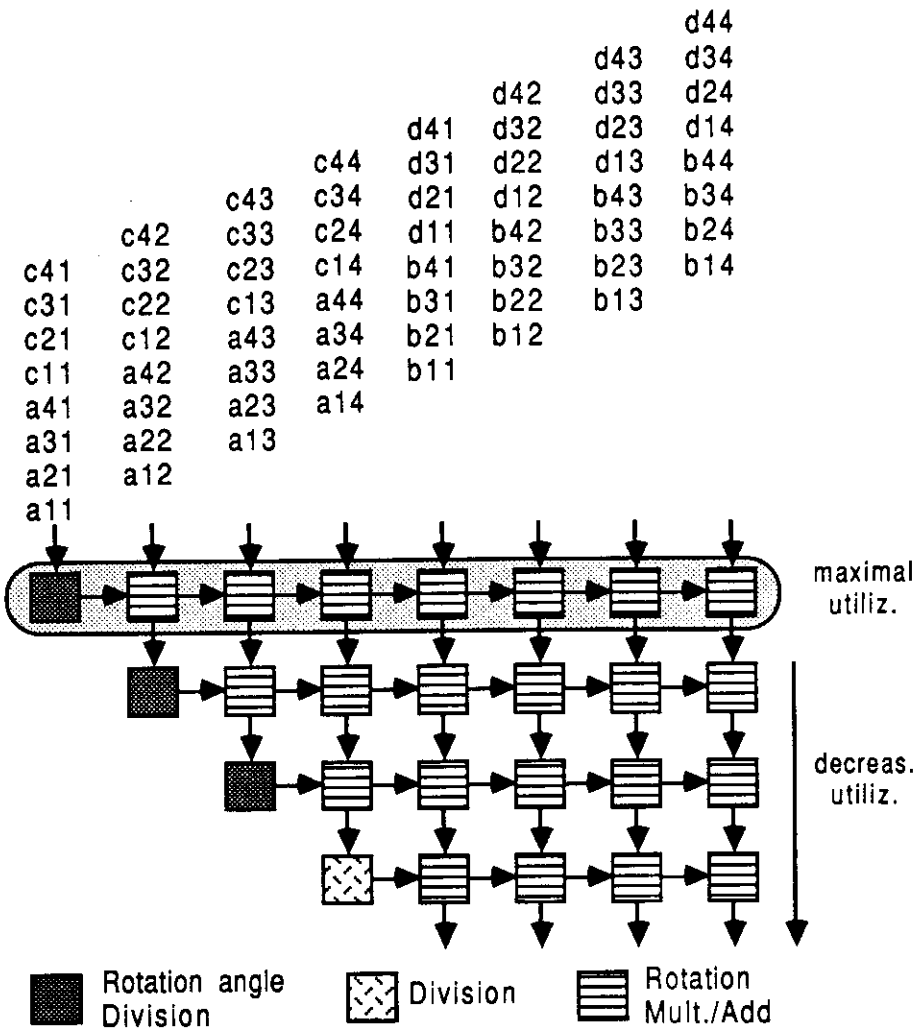


Figure 11: Trapezoidal graph from grouping vertical sub–paths

per group is not constant. Since the cell with longest computation time performs $2n$ operations, throughput of this scheme is

$$T_{\text{trapez}} = \frac{1}{2n} \quad [\text{ops}]^{-1}$$

The alternative of collapsing horizontal sub–paths of the graph into single nodes leads to the graph shown in Figure 12. This graph can also be mapped onto a trapezoidal array, with characteristics somehow similar to the previous trapezoidal structure: it also has $O(n)$ input bandwidth, but results flow horizontally through the square section of the array and become available at the rightmost column of the array. Cells are less specialized than in the previous case, because now all cells must perform a complex operation such as division or computation of rotation angles, as shown in the figure. However, fewer cells are needed because diagonal cells in the triangular part are just delay registers and do not perform any computation. Throughput of this scheme is also $[2n]^{-1}$ because the cell with longest computation time performs $2n$ operations.

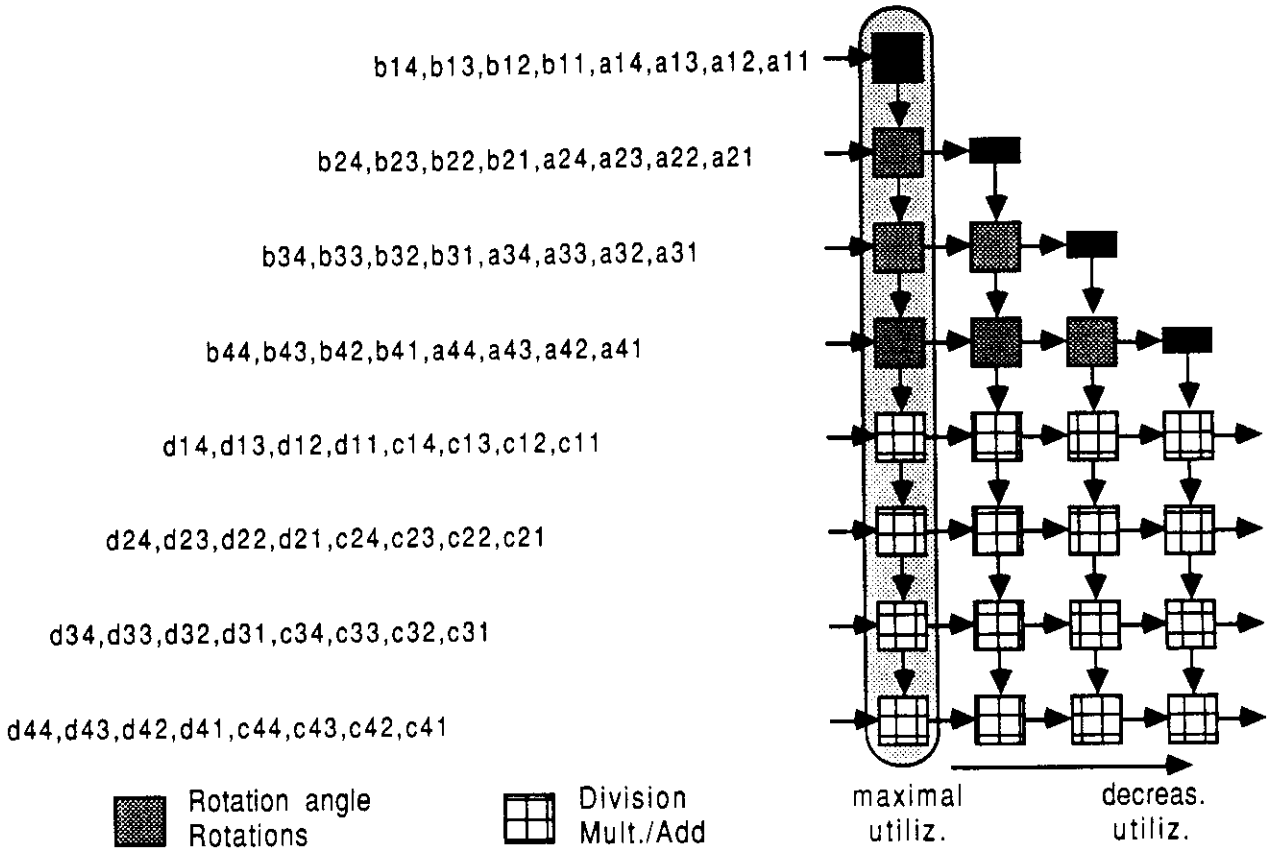


Figure 12: Trapezoidal graph from grouping horizontal sub-paths

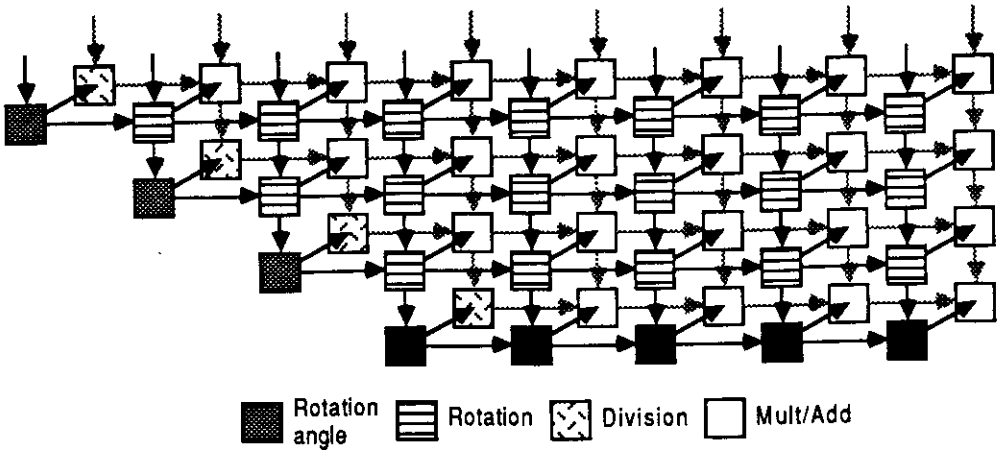


Figure 13: Pair of trapezoidal sub–graphs in parallel

In a similar manner to the case of linear structures, we can search for two–dimensional arrays with higher throughput than what is achievable in the arrays above. While grouping vertical (or horizontal) sub–paths, we can separate nodes in the different sections of the graph and group them independently. Such approach leads to pairs of trapezoidal sub–graphs as the ones described above. Nodes at the same position in these sub–graphs are from the same vertical (or horizontal) sub–path and since there is a connection along such sub–path, the two sub–graphs have their corresponding cells interconnected. Figure 13 shows the graph obtained after grouping vertical sub–paths, which can be mapped directly onto parallel trapezoidal structures. The cell with longest computation time in such scheme performs n operations, so that throughput is

$$T_{\text{dble trapez}} = \frac{1}{n} \quad [\text{ops}]^{-1}$$

3.3 Comparison of arrays for Faddeev algorithm with fixed–size matrices

In this section, we compare the characteristics of the different arrays to compute the Faddeev algorithm for fixed–size matrices devised in previous sections. Such comparison is based on information obtained from the dependence graphs of the algorithm, both the original graph shown in Figure 2 and the transformed graphs leading to the various arrays. We use the following performance measures to compare these arrays:

- Number of cells
- Throughput
- Input bandwidth
- Utilization

Number of cells and input bandwidth in each array are obtained directly from the transformed dependence graphs. Throughput is determined by the node with longest computation time in the transformed graphs. Utilization is usually computed as $\frac{\sum_i n_i t_i}{m/T}$, where n_i is the number of

nodes with computation time t_i, m is the total number of cells, and T is the throughput of the implementation. The expression $\sum_i n_i t_i$ corresponds to the total number of nodes in the original dependence graph so that utilization can be computed by using information obtained from the graph. In other words, we do not need to obtain the computation time of each cell in an array but only to compute throughput as indicated above and compute the total number of nodes in the graph.

To compute the number of nodes in Figure 2, we use the fact that such graph is composed of many rectangular sub–graphs of decreasing dimensions. Smaller rectangular sub–graphs are embedded inside larger ones. We ignore delay nodes shown in Figure 2 because those nodes do not perform useful computation. The number of nodes N is given by

$$\begin{aligned} N &= \sum_{i=0}^{n-1} (2n-i)(2n-i-1) \\ &= \frac{7}{3}n^3 - \frac{n}{3} \end{aligned}$$

The expression above is different from the one given by DeGroot et al. in [19], because they count as operations cycles when a cell is waiting to collect the first two operands before performing the first operation in a group. Such delay is due to the single–input capacity of cells. Consequently, their measure of complexity of the algorithm (i.e., $3n^3 + n^2$ operations) is greater than the actual value.

The performance measures listed above are shown in Table 1 for the arrays for fixed–size problems (the $\pm$ sign in the entries for trapezoidal structures is due to the difference in number of cells between both trapezoidal schemes). The table includes a relative measure of complexity of cells: a "simple" cell corresponds to a cell which doesn't perform division or computation of rotation angles, while a "complex" cell performs these two operations. Note that such classification is not rigorous, because it is not clear whether a cell performing computation of rotation angles is necessarily more complex than one performing rotations. Such conclusion is highly implementation–dependent, as shown in [24,25,26].

From the expressions in Table 1, we infer that utilization of linear and trapezoidal arrays is better than that of double–linear or square arrays. Among linear arrays, the one with feedback and shifting of intermediate results, shown in Figure 6, seems more advantageous due to the specialized but simpler cells. Trapezoidal arrays are attractive because they offer higher throughput than linear schemes, without degradation in utilization. Furthermore, the scheme with two–planes offers twice the throughput of the other trapezoidal arrays with the same utilization. Thus, such alternative is more attractive if one can afford the larger number of PEs and input pads.

4 The partitioning problem in the Faddeev algorithm

In the previous section we devised linear and two–dimensional arrays for the Faddeev algorithm for fixed–size matrices. However, many applications require processing large matrices for which it is not feasible to build an array of the required size. Other cases require to solve problems of variable size using the same array. As a consequence, it becomes necessary to decompose the problem into

Array	# Cells	Throughput [1/ops]	Input pads	Utilization	Cells
Linear w/regs	$2n$	$\frac{2}{3n^2-n+2}$	$2n$	$\frac{7n^2-1}{9n^2-3n+6)} \rightarrow \frac{7}{9}$	all cells complex, $(n-1)$ reg/cell
Linear w/feedback	$2n$	$\frac{2}{3n^2+n}$	$2n$	$\frac{7n^2-1}{9n^2+3n)} \rightarrow \frac{7}{9}$	all cells complex
Linear w/shift	$2n$	$\frac{2}{3n^2+n}$	$2n$	$\frac{7n^2-1}{9n^2+3n)} \rightarrow \frac{7}{9}$	one complex, $n-1$ simple
Double linear	$4n$	n^{-2}	$4n$	$\frac{7n^2-1}{12n^2} \rightarrow \frac{7}{12}$	two complex, $2(n-1)$ simple
Square no shift	$4n^2$	n^{-1}	$2n^2$	$\frac{7n^2-1}{12n^2} \rightarrow \frac{7}{12}$	$\frac{n}{2}(3n-1)$ complex, $\frac{n}{2}(5n-3)$ simple; $2n$ delays
Square w/shift	$4n^2$	n^{-1}	$2n^2$	$\frac{7n^2-1}{12n^2} \rightarrow \frac{7}{12}$	$(2n-1)$ complex, $(2n-1)^2$ simple; $2n$ delays
Trapezoidal	$\frac{3}{2}n^2 \pm \frac{n}{2}$	$[2n]^{-1}$	$2n$	$\frac{7n^2-1}{9n^2\pm3n} \rightarrow \frac{7}{9}$	n complex, $\frac{3n^2}{2} \pm \frac{n}{2}$ simple
Bi-trapezoidal	$3n^2 \pm n$	n^{-1}	$4n$	$\frac{7n^2-1}{9n^2\pm3n} \rightarrow \frac{7}{9}$	$2n$ complex, $3n^2 \pm n$ simple

Table 1: Performance measures for arrays for fixed size problems

sub–problems so that sub–problems fit into a target array. This is known as *partitioning* [27]–[30]. We first review some structures previously proposed to execute the Faddeev algorithm in partitioned mode and later present our partitioning method.

4.1 Partitioned structures previously proposed

A structure to compute the Faddeev algorithm in partitioned mode was proposed by Nash et al. in [17]. This structure is based on the one proposed in [15] for fixed–size matrices. It consists of a square array which is used to process both triangular and rectangular portions of a trapezoidal model of the algorithm such as the one shown in Figure 11. The scheme partitions matrices A and B into vertical strips as wide as the size of the array and feeds such strips sequentially to the array. After both A and B have been processed completely, matrices C and D are partitioned and processed in the same manner.

Putting Nash et al. procedure in terms of transformations as those described here, they separate the nodes in the different sections of the dependence graph shown in Figure 2 and group them independently, leading to a pair of trapezoidal sub–graphs as the one shown in Figure 13. Moreover, they map each trapezoidal sub–graph independently onto the target square array. As we described in Section 3.2, the two sub–graphs have their corresponding nodes interconnected. Such interconnections represent intermediate results from processing A and B that are needed later to process C and D. Mapping the trapezoidal sub–graphs independently implies that intermediate results are left stored inside the array. As a consequence, Nash et al. scheme requires an unloading/loading step every time a new part of a strip is brought into the array for processing. In addition, a skewing/de–skewing procedure is used for maximal utilization of the array when computing the square portions of the algorithm. The authors claim that such overhead is not significant, although they do not provide figures for such claim.

A different partitioning approach was used by Chuang and He in [30], based on I/O constraints.

They do not use Givens rotations for the annulment of matrix C in the Faddeev algorithm, but use neighbor pivoting instead [4]. Consequently, the dependence graph for their version of the Faddeev algorithm is not the one shown in Figure 2. They describe the algorithm using a trapezoidal model as the graph shown in Figure 11. Different partitioning schemes are proposed, leading to different structures. The most adequate of those structures consist of a triangular and a square array of the same width. To use such structure, they partition the trapezoidal model into horizontal strips as wide as the size of the arrays. The triangular portion of the horizontal strips is executed in the triangular array, while the remaining of the strips is executed in the square array. Since the size of the rectangular part of a strip is larger than the triangular part, the square array is used several times in each horizontal strip while the triangular array is used only once leading to low utilization of the triangular array. The other partitioning schemes proposed by Chuang and He exhibit similar characteristics.

Another work on partitioning the Faddeev algorithm was presented by De Groot et al. in [19]. They describe the implementation of a partitioned scheme that corresponds to one of those proposed by Nash et al. in [17]. It consists of visualizing the algorithm as implemented by a large virtual array and mapping multiple contiguous cells from the virtual array into each cell of the target array. Nash et al., who refer to this scheme as "coalescing," discarded it in spite of its simplicity because it requires $O(n^2)$ memory locations per cell. De Groot et al. implementation uses an array of transputers with 128K memory per processor, so that memory is not a major limitation. In fact, their main concern is to increase the processor utilization due to the low communication bandwidth of their array. Consequently, they pay the cost of increased memory requirements with the objective of reducing communications.

4.2 Partitioning procedure

We have devised an approach towards partitioning based on the dependence graph of algorithms. We briefly describe here such method and present its application to the design of arrays to compute the Faddeev algorithm in partitioned mode. We refer the reader to [31] for further details on the methodology used.

Our approach to partitioning consists of applying the following three–step procedure:

1. Transform the fully–parallel dependence graph to remove properties not suitable for an implementation, such as data broadcasting or non-regular communication pattern. Procedures for these purposes have been proposed in [21,22,23].

2. Transform the graph obtained in (1) into a new graph, which we call the *G–graph*, by collapsing groups of nodes into new nodes (*G–nodes*). The objective of this transformation is to obtain a graph more suitable for partitioning, in terms of communication requirements. An example is shown in Figure 14, where sets of consecutive nodes in diagonal sub–paths have been collapsed into *G–nodes*. Consequently, the number of nodes in the G–graph is smaller than the number of nodes in the graph used as input to this transformation.

3. Map G–nodes to a target array with m cells by scheduling sets of m neighbor G–nodes (a *G–set*) for concurrent computation, as shown in Figure 15. G-sets scheduled successively are executed in overlapped (pipelined) manner in the array. The selection of G–sets depends on

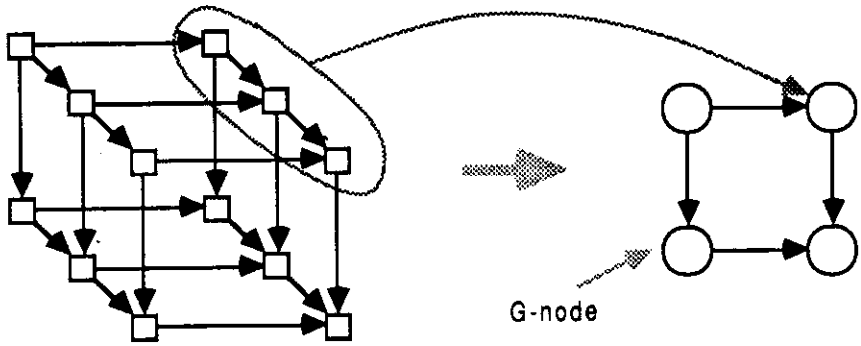


Figure 14: Collapsing primitive nodes into G–nodes

the structure of the target array. In addition, for good utilization, all nodes in a G–set should have the same computation time.

We apply now this procedure to design arrays for partitioned execution of the Faddeev algorithm.

4.3 Partitioning the Faddeev algorithm for linear arrays

Let's assume that we want to partition the Faddeev algorithm for n by n matrices so that it fits in a linear structure with only m cells, where $m \ll n$. The graph in Figure 2 has regular communications requirements among nodes, so that we do not need to perform step 1 in the procedure above. To obtain the G–graph as indicated in step 2, we consider here the case of collapsing each vertical sub–path of the graph in Figure 2 into a G–node (collapsing horizontal sub–paths is also an alternative). Grouping vertical sub–paths corresponds to the grouping done in Section 3.2 leading to the trapezoidal graph shown in Figure 11. In this graph, rows of G–nodes have the same computation time and such time decreases for lower rows of the G–graph.

In the last step of our procedure, we map G–sets from the transformed graph onto a linear array by selecting G–sets of m G–nodes in horizontal sub–graphs, as shown in Figure 16a. Scheduling of such G–sets is discussed later. Intermediate results from G–sets are saved in external memories. Those intermediate results include rotation angles or pivots flowing horizontally, and rotated or pivoted rows flowing vertically. Such data is available at the boundary of the set, so that saving it in external memories is straight–forward.

The structure resulting from the approach outlined above is shown in Figure 16b. This array enjoys maximal utilization because all G–nodes executed concurrently have the same computation time, except when executing boundary sets in some rows which might not use all cells in the array.

4.4 Partitioning the Faddeev algorithm for two–dimensional arrays

We apply now the partitioning procedure described earlier to obtain two–dimensional arrays for the Faddeev algorithm. The first two steps of such procedure are the same as for linear arrays, so

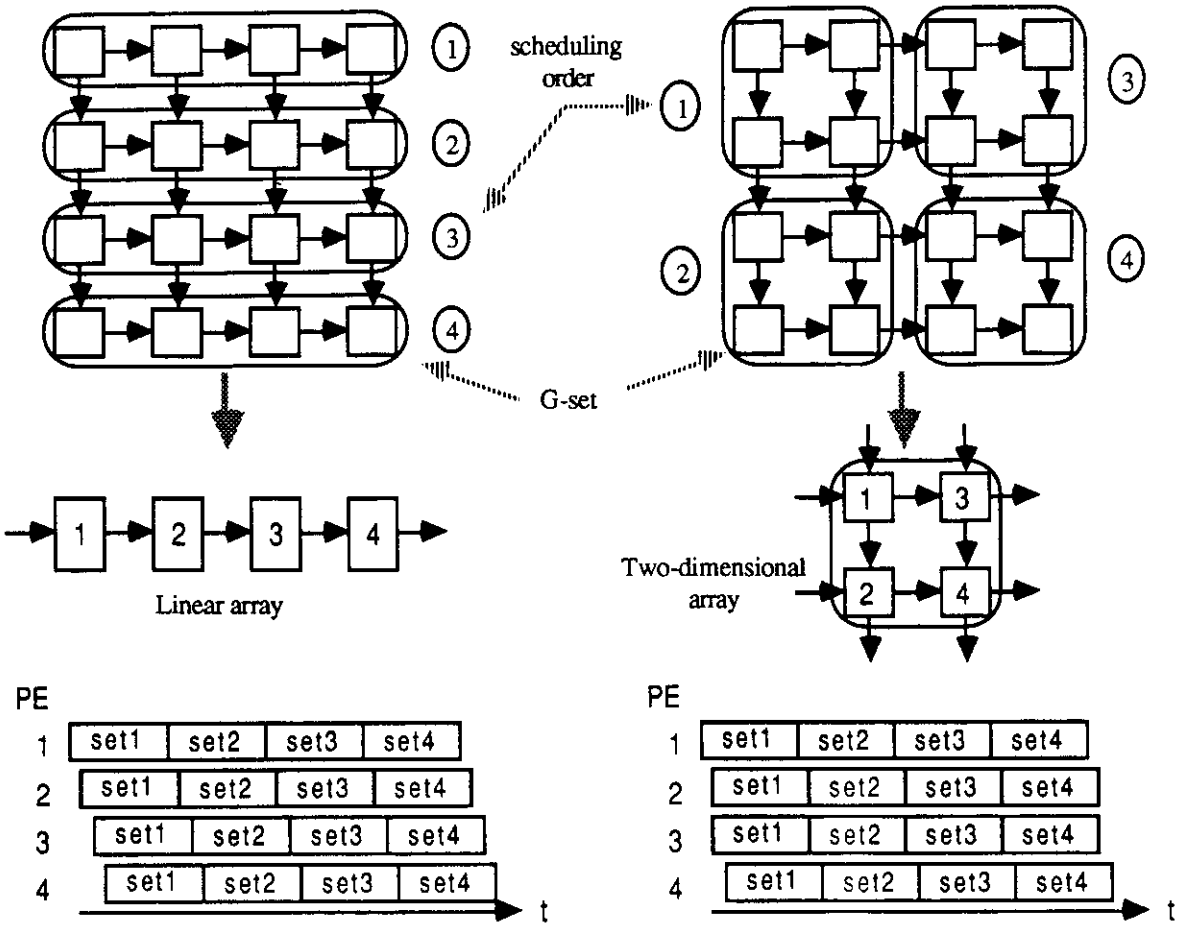


Figure 15: Mapping G–graph into linear and two–dimensional arrays

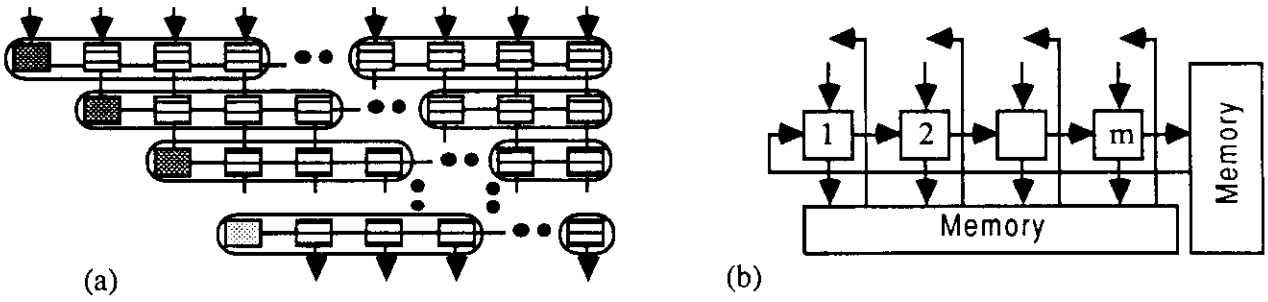


Figure 16: Partitioned linear array for the Faddeev algorithm

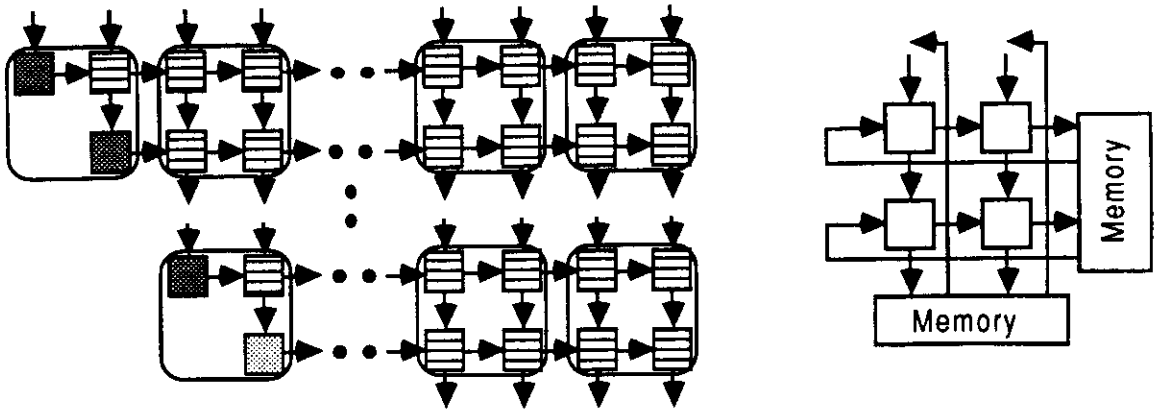


Figure 17: Two-dimensional partitioning of the Faddeev algorithm

that we use the G–graph obtained above which is shown in Figure 11.

The last step of our procedure maps the G–graph onto the target array. Mapping the trapezoidal G–graph for execution in a two–dimensional structure with m cells requires to simulate a triangular array and a square array, because those are the major components of the G–graph. Both requirements can be fulfilled in a square array, with the proper control signals. G–sets are mapped onto the array as square blocks of $\sqrt{m}$ by $\sqrt{m}$ nodes, as shown in Figure 17a. Intermediate results are saved in external memories. Those intermediate results consist of rotation angles or pivots flowing horizontally, and rotated or pivoted rows flowing vertically. The structure resulting from this approach is shown in Figure 17b. Utilization of this array is not maximal, because the computation time of G–nodes is not the same for all nodes in a G–set.

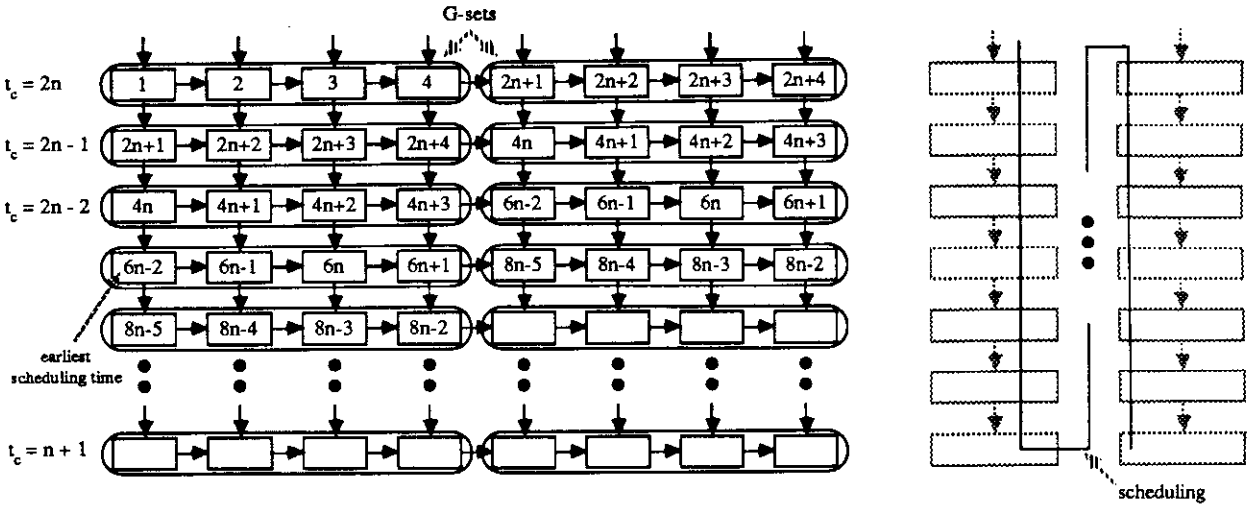
From this analysis we infer that there are differences in the amount of data that needs to be saved in external memories. The linear structure requires to save $m + 1$ data elements per G–set while the two–dimensional array requires to save only $2\sqrt{m}$.

4.5 Scheduling and I/O in partitioned Faddeev algorithm

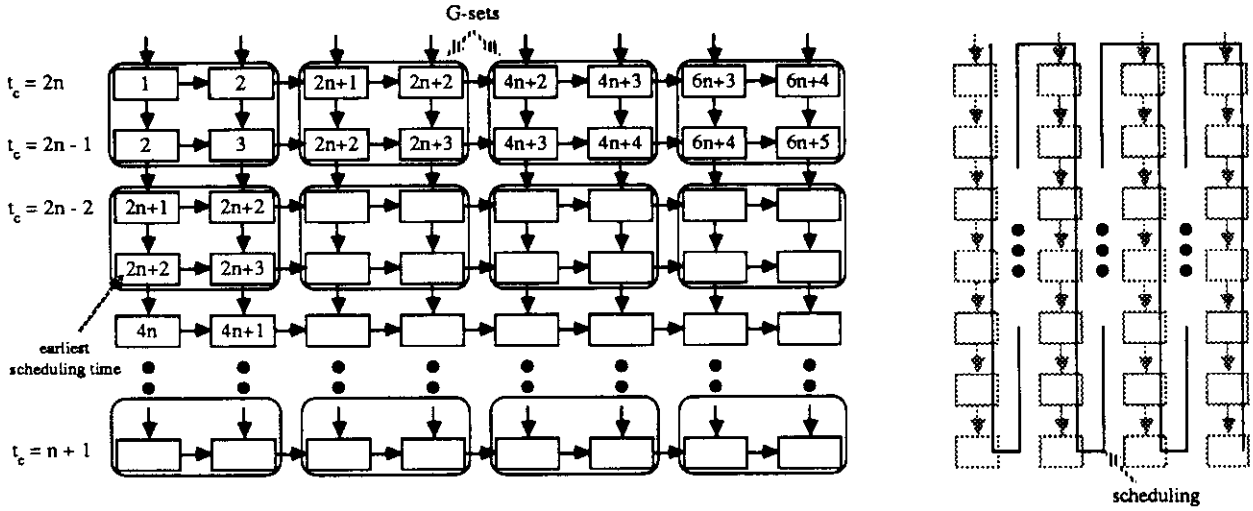
We discuss now the scheduling of G–sets mapped onto linear and two–dimensional arrays. To illustrate such scheduling, we use the G–graph shown in Figure 18 (this graph can be regarded as the internal portion of a large–size Faddeev G–graph). G–nodes in horizontal sub–paths have identical computation time and data flows in pipelined manner. Nodes in Figure 18 have been tagged with their earliest scheduling time (i.e., at what time they could start execution) relative to a reference time t_i.

Scheduling of G–sets must take into account the dependences among G–sets, due to the pipelined nature of data flow within the array. A G–set can't be scheduled for computation until required data from predecessor G–sets is available. However, computation time of nodes in a G–set is $O(n)$ while the length of dependences through the array is $O(m)$ because there are only $O(m)$ cells. Since $m \ll n$, data needed to schedule execution of a G–set is available before the G–set in execution completes. Consequently, scheduling needs to consider only the dependences between G–sets.

Mapping onto a linear array was performed by composing a G–set with nodes in horizontal



(a) - Scheduling G-graph into linear array



(b) - Scheduling G-graph into two-dimensional array

Figure 18: Scheduling G–graph into linear and two–dimensional array

sub–paths, because such nodes have the same computation time. Scheduling of G–sets can be done by horizontal or vertical sub–paths. For I/O bandwidth reasons discussed below, we choose to schedule G–sets by vertical sub–paths as depicted in Figure 18a. This figure shows that G–sets can be scheduled in pipelined mode in a simple manner. Scheduling G–sets for execution in a two–dimensional array is similar to the linear array discussed above. For I/O bandwidth reasons, we also choose to schedule G–sets by vertical sub–paths, as illustrated in Figure 18b.

A host feeding input data to the array needs to provide only the elements appearing as input to the top rows of the G–graph. Intermediate values are saved in and obtained from external memories attached to the array. Since G–sets comprising nodes at the top of the G–graph are not scheduled successively, the host needs to fed data to the array at a rate lower than one input per cell per cycle. Consequently, we can increase utilization of I/O connections by decoupling computation from data transfer, as proposed in [21,22]. Such approach leads to I/O structures as shown in Figure 19, where the host feeds data to the array through a chain of registers (the R blocks in the figure). Each block R consists of a register and a local memory with capacity to store $2n$ data elements. Data from the host flows in pipelined mode through the registers and is stored in the memories. When a G–set from the top of the graph is scheduled for execution, data is read from the memories into the PEs while new data is transferred from the host.

G–nodes in Figure 19 have been tagged with their scheduling time, as determined above. Since each node at the top of the G–graph receives $2n$ data elements from the host, I/O bandwidth is given by

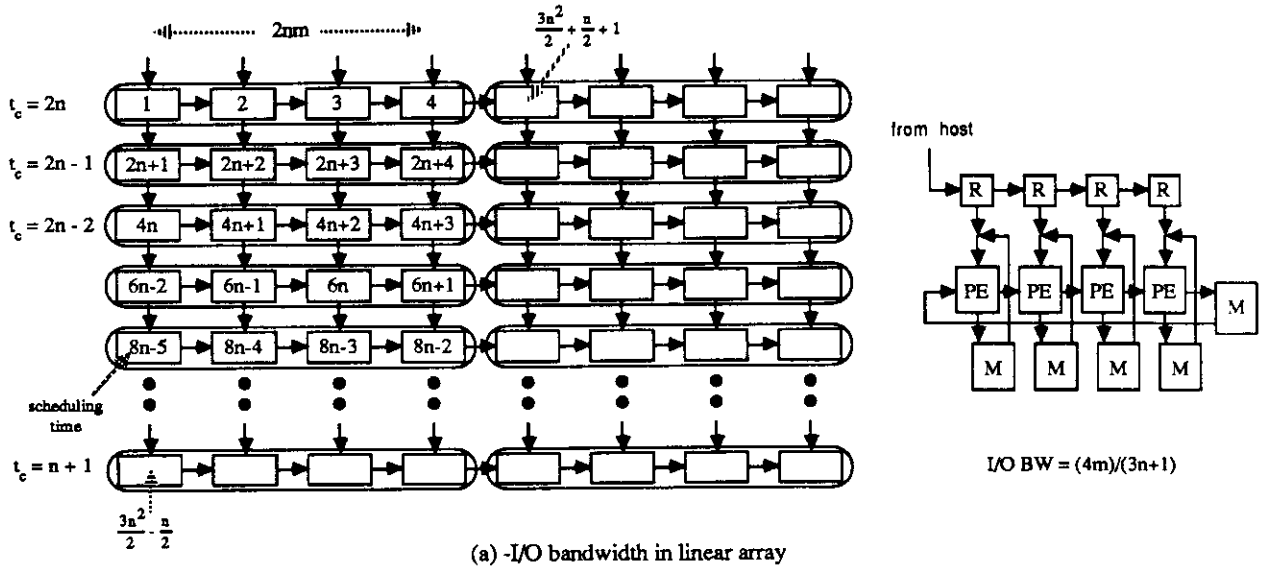
$$\begin{aligned} \text{BW} &= \frac{2nm}{\sum_{k=1}^{n} t_{c_k}} \\ &= \frac{2nm}{(2n+1)n - \frac{1}{2}n(n+1)} \\ &= \frac{4m}{3n+1} \end{aligned}$$

where t_{c_k} is computation time of G–nodes in the k-th row of the G–graph. Under the conditions described above, *linear and two–dimensional arrays have the same I/O bandwidth from the host.*

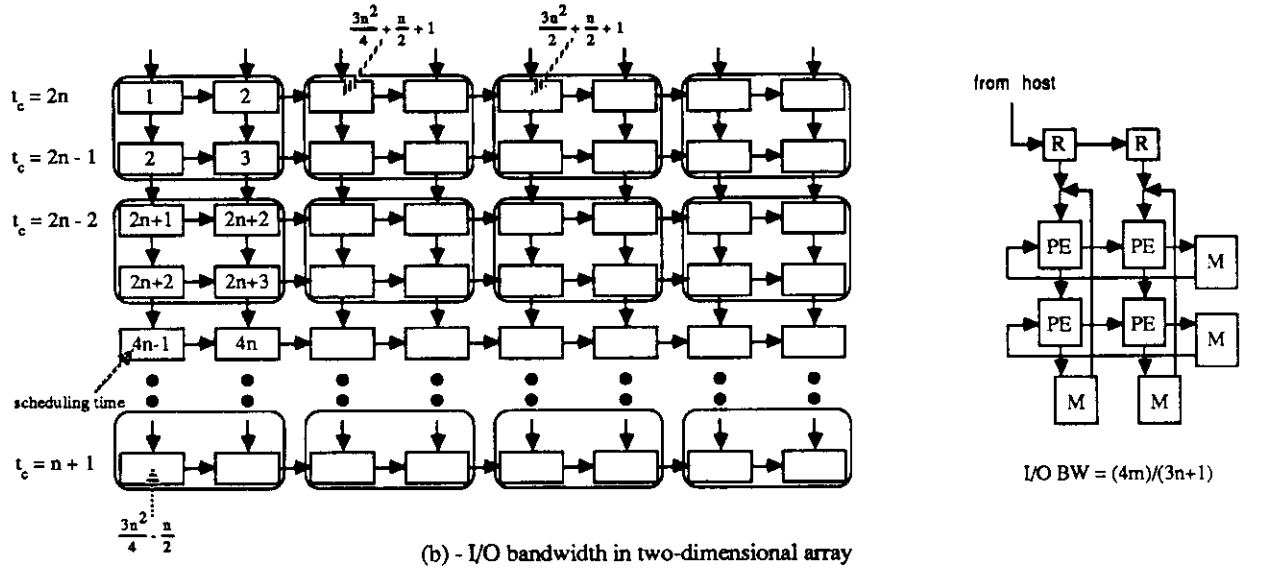
4.6 Comparison of arrays for partitioned Faddeev algorithm

We compare now the characteristics of the arrays for the Faddeev algorithm in partitioned mode presented in previous sub–sections. We use the same performance measures to compare these arrays as the ones used for fixed–size cases, plus overhead due to partitioning.

Utilization of arrays is computed in the same manner as in fixed–size problems, namely as number of nodes in the dependence graph divided by m/T, where T is throughput. As stated in Section 3.3, the number of nodes in the dependence graph is $\frac{7}{3}n^3 - \frac{n}{3}$. Throughput is determined by the computation time of the busiest cell in the array. Such information is obtained from mapping the G–graph onto the target array. In the following subsections, we present the derivation of the corresponding expressions for the different arrays.



(a) -I/O bandwidth in linear array



(b) - I/O bandwidth in two-dimensional array

Figure 19: I/O bandwidth in partitioning Faddeev algorithm

Linear array

When scheduling the G–graph shown in Figure 11 for execution in a linear array with m cells, the first horizontal sub–path of the graph is mapped in $2n/m$ sets. Each G–node in this sub–path consists of $2n$ operations. The second horizontal sub–path is mapped in $\lceil (2n-1)/m \rceil$ sets because the length of the sub–path is shorter, and each G–node consists of $2n-1$ operations. This pattern repeats for all horizontal sub–paths and the last one is mapped in $\lceil (n+1)/m \rceil$ sets. Therefore, the array is used for

$$\begin{aligned}
\sum_{i=0}^{n-1} \left\lceil \frac{2n-i}{m} \right\rceil (2n-i) &= \sum_{k=0}^{\frac{n}{m}-1} \left[\left(\frac{2n}{m} - k \right) \sum_{i=0}^{m-1} (2n-i-km) \right] \\
&= \sum_{k=0}^{\frac{n}{m}-1} \left(\frac{2n}{m} - k \right) \left[(2n-km)m - \sum_{i=0}^{m-1} i \right] \\
&= \sum_{z=\frac{n}{m}+1}^{\frac{2n}{m}} \left[m^2 z^2 - \frac{m(m-1)}{2} z \right] \qquad z = \frac{2n}{m} - k \\
&\approx \frac{28n^3 - 9n^2(m-1)}{12m} \qquad [\text{ops}]
\end{aligned}$$

and throughput is

$$T_{linear} = \frac{12m}{28n^3 - 9n^2(m-1)} \qquad [\text{ops}]^{-1}$$

Utilization is given by

$$\begin{aligned}
U_{linear} = \frac{\sum \text{nodes}}{m/T} &= \frac{\frac{1}{3}n(7n^2-1)}{m\left(\frac{1}{12m}[28n^3 - 9n^2(m-1)]\right)} \\
&= \frac{28n^2 - 4}{28n^2 - 9n(m-1)}
\end{aligned}$$

Therefore, for large n, utilization tends to 1 and throughput tends to $\frac{3}{7}\frac{m}{n^3}$.

Square array

In the square array proposed here, G–sets are scheduled as square blocks of $\sqrt{m}$ by $\sqrt{m}$ nodes. Therefore, the first $\sqrt{m}$ horizontal sub–paths of the G–graph are mapped to the cells in $2n/\sqrt{m}$ sets. Computation time of these sets is given by the computation time of nodes in the first horizontal sub–path, which consist of $2n$ operations. The next $\sqrt{m}$ horizontal sub–paths are mapped to the array in $(2n-\sqrt{m})/\sqrt{m}$ sets, because the length of these sub–paths is shorter than previous ones. Computing these sets requires $2n-\sqrt{m}$ operations. The remaining horizontal sub–paths follow a similar pattern, so that the array is used for

$$\begin{aligned}
\sum_{i=0}^{p-1} \left(\frac{2n - i\sqrt{m}}{\sqrt{m}}\right)(2n - i\sqrt{m}) &= \frac{1}{\sqrt{m}} \sum_{i=0}^{p-1} (2n - i\sqrt{m})^2 \\
&= \sqrt{m} \sum_{i=0}^{p-1} \left(\frac{2n}{\sqrt{m}} - i \right)^2
\end{aligned}$$

$$= \sqrt{m} \sum_{z=\frac{n}{\sqrt{m}}+1}^{\frac{2n}{\sqrt{m}}} z^2 \qquad z = \left(\frac{2n}{\sqrt{m}} - i\right)$$

$$\approx \frac{7}{3}\frac{n^3}{m} \quad \text{[ops]}$$

where $p = n/\sqrt{m}$ is the number of rows of $\sqrt{m}$ by $\sqrt{m}$ sets of G–nodes needed to cover the entire G–graph. Throughput is

$$T_{square} = \frac{3m}{7n^3} \quad \text{[ops]}^{-1}$$

and utilization is given by

$$U_{square} = \frac{\sum \text{nodes}}{m/T} = \frac{\frac{1}{3}n(7n^2-1)}{m\left(\frac{1}{3m}(7n^3)\right)} = \frac{7n^3 - n}{7n^3}$$

Therefore, for large n, utilization tends to 1 and throughput tends to $\frac{3}{7}\frac{m}{n^3}$

Nash et al. array

Nash et al. [17] use a square array to map their bi–trapezoidal model. They map each of the trapezoidal sub–graphs independently, so that they require certain overhead in unloading/loading and skewing/de–skewing data. Since such overhead has not been reported quantitatively, we compute the throughput of their scheme ignoring such overhead. Consequently, this is an upper bound of what is achievable with their implementation.

Computing how long their array is used can be decomposed into computing how long each of the trapezoidal sub–graphs uses the array, as follows (where $p = n/\sqrt{m}$):

- Triangularizing matrix A and transforming matrix B (i.e., executing the first trapezoidal sub–graph). In such sub–graph there are as many nodes as in the trapezoidal graph mapped onto the square array discussed above, but the computation time of each node is shorter. Consequently, execution of these G–sets takes

$$\sum_{i=0}^{p-1} \frac{2n - i\sqrt{m}}{\sqrt{m}}(n - i\sqrt{m}) = \frac{5n^3 + 6n^2\sqrt{m} + nm}{6m}$$

- Annulling matrix C (i.e., executing the triangular portion of the second trapezoidal sub–graph). Execution of these G–sets takes

$$\sum_{i=0}^{p-1} \left(\frac{n - i\sqrt{m}}{\sqrt{m}}\right) n = \frac{n^3 + n^2\sqrt{m}}{2m}$$

- Updating matrix D (i.e., executing the rectangular portion of the second trapezoidal sub–graph). Such operation takes

$$\sum_{i=0}^{p-1} \left(\frac{n}{\sqrt{m}}\right) n = \frac{n^3}{m}$$

These three terms together with an extra term accounting for overhead in data transfers give the throughput of this implementation as

$$T_{\text{Nash}} = \frac{6m}{14n^3 + 9n^2\sqrt{m} + nm + 6m(\text{OVHD})} \quad [\text{ops}]^{-1}$$

Utilization is given by

$$U_{\text{Nash}} = \frac{\sum \text{nodes}}{m/T} = \frac{14n^2 - 2}{14n^3 + 9n^2\sqrt{m} + nm + 6m(\text{OVHD})}$$

Consequently, Nash et al. implementation has the same throughput as the square array proposed above if there is no overhead in data transfers. In practice, such throughput and utilization of the array are lower. In addition, their scheme exhibits complexity in the control required to perform those data transfers into and out of the array. I/O bandwidth of Nash et al. scheme is higher than the square array above, because of the loading/unloading of data.

De Groot et al. partitioned scheme

De Groot et al. [19] evaluate their partitioning scheme and array considering that data communication is ten times slower than performing a single operation in a cell. This is a consequence of their implementation, an hypercube with transputers as nodes. In addition, they use completion time as performance measure. Although completion time is an important parameter for certain applications, we believe that throughput in an array is more important so we use the latter for our evaluation.

De Groot's scheme consists of a trapezoidal structure with $(3P^2 + P)/2$ cells, where P is the dimension of the square and triangular portions of their array. To compare such array with the ones derived here, we express P in terms of m, that is in terms of the number of cells in our arrays. Thus $(3P^2 + P)/2 = m$ leads us to $P = \frac{1}{6}[\sqrt{24m + 1} - 1]$. The trapezoidal model describing the algorithm is partitioned into blocks of adjacent nodes and each block is mapped onto a single cell. Consequently, in the square portion of the trapezoidal model such blocks have (n/P) vertical sub–paths with (n/P) nodes in each sub–path. Throughput of their implementation is given by the computation time of the first row of cells in the array, since those cells have the most operations to compute. Such cells perform $2n + (2n-1) + \cdots + (2n - \frac{n}{P} + 1)$ operations for each of the (n/P) vertical sub–paths. Thus, computation time of the cells is

$$\begin{aligned} t_{cell} &= \frac{n}{P}\left[2n + (2n-1) + \cdots + (2n - \frac{n}{P} + 1)\right] \\ &= \frac{n}{P}\sum_{i=1}^{n/P}(2n - \frac{n}{P} + i) \\ &= \frac{n}{P}\left[\sum_{z=2n-\frac{n}{P}+1}^{2n} z\right] \qquad z = 2n - \frac{n}{P} + i \\ &\approx 2n\left(\frac{n}{P}\right)^2 - \frac{1}{2}\left(\frac{n}{P}\right)^3 \quad [\text{ops}] \end{aligned}$$

Replacing the value of P computed above, we obtain

$$\begin{aligned} t_{cell} &= \frac{36n^3}{12m - \sqrt{24m+1}+1} - \frac{54n^3}{(12m - \sqrt{24m+1}+1)(\sqrt{24m+1}-1)} \\ &= \frac{18n^3}{12m - \sqrt{24m+1}+1}\left[2 - \frac{3}{\sqrt{24m+1}-1}\right] \\ &\approx \frac{36n^3}{12m - \sqrt{24m+1}+1} \quad [\text{ops}] \end{aligned}$$

so that throughput is

$$T_{DeGroot} = \frac{12m - \sqrt{24m+1}+1}{36n^3} \quad [\text{ops}]^{-1}$$

and utilization becomes

$$\begin{aligned} U_{DeGroot} = \frac{\sum \text{nodes}}{N/T} &= \frac{\frac{1}{3}n(7n^2-1)}{m\left(\frac{36n^3}{12m-\sqrt{24m+1}+1}\right)} \\ &= \frac{(7n^2-1)(12m - \sqrt{24m+1}+1)}{108n^2m} \\ &= \frac{84n^2m - 12m + (7n^2-1)(1-\sqrt{24m+1})}{108n^2m} \end{aligned}$$

Therefore, for large n, utilization tends only to 7/9 and throughput tends to $\frac{1}{3}\frac{m}{n^3}$.

The results above are summarized in Table 2. We haven't completed the entries in the table for Nash et al. array, because the overhead in loading/skewing data has not been reported quantitatively. Furthermore, we have not included I/O bandwidth for De Groot et el. scheme, because of their approach towards data transfer. From this table we infer that, for large n, both our linear and square arrays tend to the same throughput (i.e., $\frac{3}{7}\frac{m}{n^3}$) and optimal utilization. In addition, both exhibit the same I/O bandwidth from the host. These linear and square arrays have better performance measures than the array proposed by De Groot et al. and do not exhibit the overhead required in the scheme proposed by Nash et al.

In addition to the performance measures described above, a linear array is more advantageous than a two–dimensional one because:

- it is simpler to implement
- for a finite value of n it has slightly higher utilization than the two–dimensional structure
- it is better suited to incorporate fault–tolerant capabilities (i.e., it's easier to skip a faulty cell in a linear array than to reconfigure a two–dimensional structure)

Consequently, we conclude that *for partitioned execution of the Faddeev algorithm, a linear array offers better performance and implementation than a two–dimensional array.*

5 Conclusions

We have presented the application of a graph–based methodology to derive arrays for computing the Faddeev algorithm, for fixed–size and partitioned problems. We have derived linear and two–dimensional implementations for such algorithm, and we have compared these arrays with other

Table 2: Performance measures for partitioned implementations with m cells

Array	Throughput [1/ops]	I/O BW	Utilization	Overhead
Linear	$\frac{12m}{28n^3-9n^2(m-1)}$	$\frac{4m}{3n+1}$	$\frac{28n^2-4}{28n^2-9n(m-1)} \rightarrow 1$	none
Square	$\frac{3m}{7n^3}$	$\frac{4m}{3n+1}$	$\frac{7n^3-n}{7n^3} \rightarrow 1$	none
De Groot	$\frac{12m-\sqrt{24m+1}+1}{36n^3}$	—	$\frac{(7n^2-1)(12m-\sqrt{24m+1}+1)}{108n^2m}$ $\rightarrow 7/9$	$O(n^2)$ storage
Nash	$\frac{6m}{14n^3+9n^2\sqrt{m}+nm+6m(\text{ovhd})}$	$\frac{4m}{3n+1}$ + ovhd	$\frac{14n^2-2}{14n^3+9n^2\sqrt{m}+nm+6m(\text{ovhd})}$	loading, skewing

schemes previously proposed in the literature. The evaluation used performance measures such as number of processing elements (PEs), throughput, I/O bandwidth, utilization of PEs and overhead due to partitioning.

Our results show that, for fixed–size problems, throughput reaches $2/(3n^2 - n + 2)$ in a linear array with $2n$ cells and $n-1$ storage locations per cell. Other linear schemes proposed here achieve throughput $2/(3n^2+n)$ also with $2n$ cells but no storage requirements in each cell. Two–dimensional schemes reach throughput $(2n)^{-1}$ or n^{-1} with $2n$ and $4n$ cells respectively. Utilization of all these arrays tends to 7/9. One of the two–dimensional schemes presented here corresponds to the one proposed in [15].

For partitioned problems, we mapped the algorithm onto linear and two–dimensional structures. We derived a two–dimensional scheme that is more efficient and has less overhead than other arrays previously proposed. We have shown that throughput of our partitioned implementation, both linear and two–dimensional, tends to $(3m)/(7n^3)$, where m is the number of cells. Moreover, we have shown that both linear and two–dimensional structures have the same I/O bandwidth from the host, namely $(4m)/(3n + 1)$ [words/ops].

We have shown that performance measures of partitioned implementations of the Faddeev algorithm are the same for both linear and two–dimensional arrays, but linear schemes are simpler and better suited to incorporate fault–tolerant properties. Consequently, we concluded that, for partitioned execution of the Faddeev algorithm, a linear array offers better performance and implementation than a two–dimensional array.

References

[1] H. Kung, "Let's design algorithms for VLSI systems," in *CALTECH Conference on VLSI*. pp. 65–90, 1979.

[2] H. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, pp. 37–46, Jan. 1982.

[3] H. Ahmed, J. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *IEEE Computer*, vol. 15, pp. 65–82, Jan. 1982.

[4] W. Gentleman and H. Kung, "Matrix triangularization by systolic arrays," in *SPIE Real-Time Signal Processing IV*, pp. 19–26, 1981.

[5] J. Speiser and H. Whitehouse, "A review of signal processing with systolic arrays," in *SPIE Real-Time Signal Processing VI*, pp. 2–6, 1983.

[6] D. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proceedings of the IEEE*, vol. 71, pp. 113–120, Jan. 1983.

[7] F. Luk, "Architectures for computing eigenvalues and SVD's," in *SPIE Highly Parallel Signal Processing Architectures*, pp. 24–33, 1986.

[8] A. Fisher, H. Kung, and L. Monier, "Architecture of the PSC: a programmable systolic chip," in *10th Annual Symposium on Computer Architecture*, pp. 48–53, 1983.

[9] L. Snyder, "Introduction to the configurable highly parallel machine," *IEEE Computer*, vol. 15, pp. 47–64, Jan. 1982.

[10] J. Symanski, "Implementation of matrix operations on the two-dimensional systolic array testbed," in *SPIE Real-Time Signal Processing VI*, pp. 136–142, 1983.

[11] B. Drake, F. Luk, J. Speiser, and J. Symanski, "SLAPP: a systolic linear algebra parallel processor," *IEEE Computer*, vol. 20, pp. 45–50, July 1987.

[12] M. Annaratone, E. Arnould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, , and J. Webb, "The Warp computer: architecture, implementation and performance," *IEEE Transactions on Computers*, vol. C-36, pp. 1523–1538, Dec. 1987.

[13] D. Foulser and R. Schreiber, "The Saxpy Matrix-1: a general purpose systolic computer," *IEEE Computer*, vol. 20, pp. 35–44, July 1987.

[14] D. Faddeev and V. Faddeeva, *Computational Methods of Linear Algebra*, pp. 150–158. W.H. Freeman and Co., 1963.

[15] J. Nash and S. Hansen, "Modified Faddeev algorithm for matrix manipulation," in *SPIE Real-Time Signal Processing VII*, pp. 39–46, 1984.

[16] J. Nash, K. Przytula, and S. Hansen, "Systolic/cellular processor for linear algebraic operations," in *VLSI Signal Processing II*, (J. N. S.Y. Kung, R. Owen, ed.), pp. 306–315, IEEE Press, 1986.

[17] J. Nash, S. Hansen, and K. Przytula, "Systolic partitioned and banded linear algebraic computations," in *SPIE Real-Time Signal Processing IX*, pp. 10–16, 1986.

[18] H. Chuang and G. He, "A versatile systolic array for matrix computations," in *12th Annual Symposium on Computer Architecture*, pp. 315–322, 1985.

[19] A. DeGroot, E. Johansson, and S. Parker, "Systolic array for efficient execution of the Faddeev algorithm," in *SPIE Real-Time Signal Processing X*, pp. 86–93, 1987.

[20] J. Fortes, K. Fu, and B. Wah, "Systematic approaches to the design of algorithmically specified systolic arrays," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 300–303, 1985.

[21] J. Moreno, "A proposal for the systematic design of arrays for matrix computations," Technical Report CSD–870019, Computer Science Department, University of California Los Angeles, May 1987.

[22] J. Moreno and T. Lang, "Design of special–purpose arrays for matrix computations: preliminary results," in *SPIE Real–Time Signal Processing X*, pp. 53–65, 1987.

[23] J. Moreno and T. Lang, "Reducing the number of cells in arrays for matrix computations," Technical Report, Computer Science Department, University of California Los Angeles, March 1988.

[24] M. Ercegovac and T. Lang, "Redundant and on-line CORDIC: application to matrix triangularization and SVD," Technical Report CSD–870046, Computer Science Department, University of California Los Angeles, 1987.

[25] M. Ercegovac and T. Lang, "On-line scheme for computing rotation factors," in *8th Symposium on Computer Arithmetic*, pp. 196–203, 1987.

[26] J. Cavallaro and F. Luk, "CORDIC arithmetic for an SVD processor," in *8th Symposium on Computer Arithmetic*, pp. 215–222, 1987.

[27] K. Hwang and Y. Cheng, "Partitioned matrix algorithms for VLSI arithmetic systems," *IEEE Transactions on Computers*, vol. C–31, pp. 1215–1224, Dec. 1982.

[28] J. Navarro, J. Llaberia, and M. Valero, "Partitioning: an essential step in mapping algorithms into systolic array processors," *IEEE Computer*, vol. 20, pp. 77–89, July 1987.

[29] D. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, Jan. 1986.

[30] H. Chuang and G. He, "Design of problem–size independent systolic array systems," in *International Conference on Computer Design*, pp. 152–157, 1984.

[31] J. Moreno and T. Lang, "Graph–based partitioning of matrix algorithms for systolic arrays," Technical Report, Computer Science Department, University of California Los Angeles, March 1988.