GRAPH-BASED PARTITIONING OF MATRIX ALGORITHMS FOR SYSTOLIC ARRAYS

Jaime H. Moreno
Tomas Lang

March 1988
CSD-880015

Graph-based Partitioning of Matrix Algorithms for Systolic Arrays

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 propose a technique to partition algorithms for execution in systolic arrays, based on the dependence graph of algorithms. Such procedure achieves its objectives by performing transformations on the graph. These transformations first remove from the graph properties not suitable for an implementation, then transform the graph into one with simple communication requirements, and finally map the graph onto the target array. We illustrate this method through its application to the computation of transitive closure of a directed graph. We derive linear and two-dimensional structures for such algorithm that exhibit maximal utilization, no overhead due to partitioning and simple control. In the process, we obtain a graph suitable for an array for fixed-size problems that exhibits better characteristics than arrays previously proposed for this algorithm. Our method also allows evaluating trade-offs among implementations. We show that, in partitioned implementations with the same number of cells, a linear array performs better, its implementation is easier and it 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]–[6]. Many applications require processing large matrices for which it is not feasible to build an array of the required size, while others require solving problems of variable size using the same array. In such cases, it becomes necessary to decompose the problem into sub–problems so that the sub–problems fit into a target array. This is known as *partitioning* the algorithm and has been studied by many researchers [7]–[14].

In this paper, we present a partitioning technique based on the dependence graph of algorithms. This is a transformational approach, that uses a fully–parallel dependence graph as the description of the algorithm. Such a graph is first transformed to remove properties not desirable for an implementation (i.e., data broadcasting, bi–directional data flow) and converted into a graph suitable for partitioning (i.e., with simple communication requirements). The resulting graph is mapped onto the target array. The transformations are performed taking into account issues such as I/O bandwidth, throughput, delay, and utilization of PEs. We illustrate the technique through its application to the design of arrays for partitioned computation of the transitive closure of a directed graph. We derive and evaluate linear and two–dimensional structures to compute such algorithm. These arrays exhibit maximal utilization, no overhead and simple control. In addition, we show that an intermediate graph used by the methodology is suitable for implementation of fixed–size arrays for transitive closure, with better characteristics than arrays previously proposed for such computation.

We use a fully–parallel graph to describe an algorithm because such graph is unique for a given algorithm, exhibits the intrinsic properties of the algorithm, and doesn't restrict the algorithm to be of a certain class (i.e., expressed as recurrence equations or nested loops). In such a graph, nodes represent operations and edges correspond to data communications. These graphs are characterized by having all inputs and outputs available in parallel and no loops (i.e., loops are unfolded). For simplicity, we assume that computation time is the same for all operations. The validity of such an assumption is highly implementation–dependent, as suggested by studies about the design of special–purpose cells [15,16,17]. The graph can be used directly to derive an implementation by assigning each node to a different processing element (PE), and by adding delay registers to synchronize the arrival of data to the PEs (i.e., *a pipelined implementation of the graph*). The resulting structure exhibits minimum delay (determined by the longest path in the graph) and maximum throughput (determined by the computation time of the nodes). However, for a large–size problem such structure requires many units, complex interconnection and high I/O bandwidth. Our method deals with these issues, while still attempting to preserve the delay and throughput properties inherent in the dependence graph.

We have applied our partitioning technique to several algorithms for matrix computations, among them LU–decomposition, QR–decomposition, and Faddeev algorithm. Our results show that the graphical nature of our approach makes it easier to use than methodologies based on mathematical expressions proposed in the literature. Moreover, the method allows evaluating trade–offs between linear and two–dimensional arrays for partitioned execution of algorithms. This technique is an extension to one for the design of arrays for fixed–size problems that we have previously proposed [18]–[20].

Several approaches to partition algorithms for execution in arrays have been reported in the

literature. These methods are characterized by properties such as how the algorithm is represented, how partitioning is performed, and the generality of the approach. For instance, Navarro et al. [7] transform an algorithm with large–size dense matrices into an algorithm with band matrices and compute the resulting algorithm in an array that matches the size of the band. In this method, the original algorithm is decomposed into subalgorithms that are chained for execution in the target array. Such decomposition depends on the algorithm and consequently might be different from one algorithm to another, if at all possible. Moldovan and Fortes [8] use a mathematical approach for the class of algorithms representable with nested loops, in which computations are almost identical over the entire index space. They divide the index space into bands and map the bands into the processor space, in such a way that only near neighbor communications are required for index points inside the bands. The mathematical nature of this scheme allows for powerful transformations, but such transformations are complex and too distant from an implementation so that they are not easy to select. To aid in such selection, Moldovan and Fortes have used a software package that finds many valid transformations and picks one suitable for a particular objective. Kung [9] also uses a mathematical approach, based on the selection of global scheduling vectors that do not violate data dependencies of an algorithm. Kung's scheme works only for algorithms with uni–directional data dependences (i.e., no reverse data dependences).

Partitioning the computation of transitive closure of a directed graph has been recently addressed by Núñez and Torralba [22]. They propose an algorithm and partition it through decomposition into a block–algorithm. Such decomposition is dependent on the algorithm and corresponds to the class of schemes described in [7]. They show that sub–algorithms composing the computation of transitive closure correspond to sequences of matrix multiplications. Although they do not address the details of an implementation, their algorithm requires rather complex control to chain the different sub–problems.

In the next section, we present our partitioning technique and a three–step procedure to carry it out. In Section 3 we show the application of our approach to partition the computation of transitive closure of a directed graph. In Section 4 we describe our performance measures, evaluate partitioned implementations and discuss tradeoffs in the design of linear and two–dimensional arrays.

2 Graph–based partitioning

Partitioning consists of mapping the computation of an algorithm with large–size data onto an array smaller than the size of the data. Three basic approaches have been proposed to achieve such mapping:

1. Partition the algorithm into a *number of components equal to the number of PE's available in the target array.* Such components have data dependences (i.e., communication requirements) that match the interconnection structure of the array and each component is mapped onto one cell of the array. Figure 1 depicts this technique. Such figure shows a dependence graph as a rhombus, which can be partitioned into a number of communicating components that are mapped onto the array.

   This type of partitioning has received different names in the literature, such as *locally sequential globally parallel (LSGP)* [23] or the more intuitive of *coalescing* [7,11]. In this scheme, each cell sequentially executes the operations in its component, according to a certain sched-

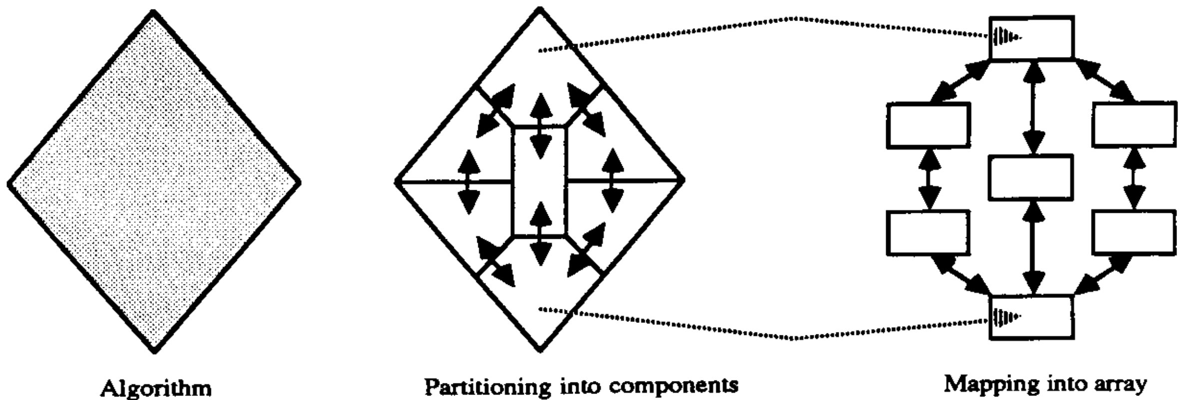


Figure 1: Partitioning an algorithm through coalescing

uling. The scheme is attractive due to its simplicity but requires local storage within each cell. Depending on the algorithm and on the partitioning, such storage requirements might be large (i.e., $O(n)$ or $O(n^2)$).

2. Partition the algorithm into *components that have computation and communication requirements matching the size and interconnection structure of the target array* and map each component onto the entire array. The different components are mapped sequentially onto the array, according to certain scheduling. Figure 2 illustrates this scheme, where a dependence graph is divided into a number of components which exhibit communication requirements in a triangular pattern. Consequently, such components are mapped onto a triangular array.

   This type of partitioning has been referred to as *locally parallel globally sequential (LPGS)* [23] or with the more intuitive name of *cut-and-pile* [7]. Depending on algorithm and partitioning, data from the components needs to be saved in memory external to the array and fed back to the input as needed.

3. Decompose the algorithm into subalgorithms that fit into the target array. The subalgorithms, which need not be the same as the original one, are executed sequentially in the array according to certain scheduling. Consequently, this approach transforms the original algorithm into a series of subproblems which, when executed, provide the same result as the original algorithm.

   An example of this approach is the one proposed by Navarro et al. [7] and shown in Figure 3. In this case, an algorithm with large size dense matrices is transformed into an algorithm with band matrices and computed in an array tailored to the band size.

These basic approaches can be combined to provide more alternatives for an implementation. For example, one could conceive a scheme where cut-and-pile is performed first to obtain partitions larger than the target array size and then coalescing is applied over the partitions. Such scheme would help reducing the memory requirements of applying coalescing alone.

To use any of the approaches above it is necessary to devise procedures to partition an algorithm into components (or decomposition into subalgorithms) and to schedule those components (or

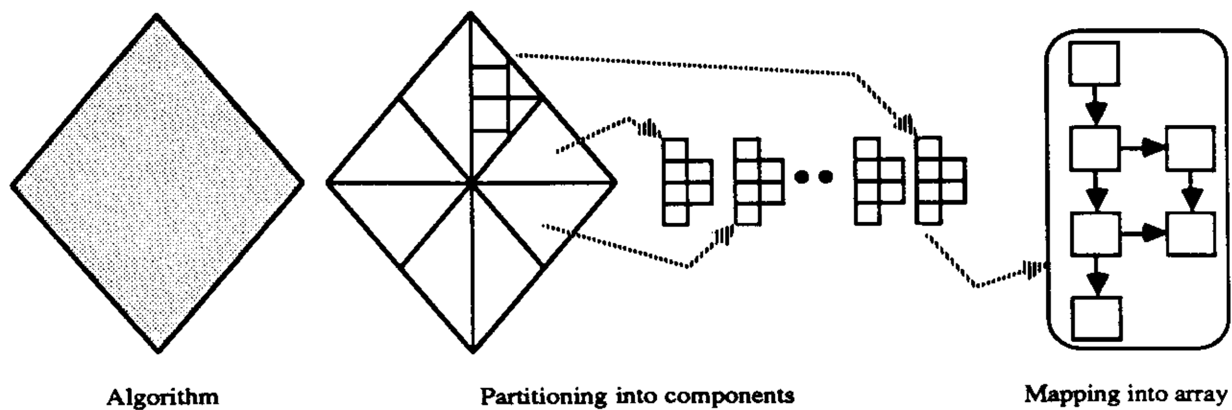


Figure 2: Partitioning an algorithm through cut–and–pile

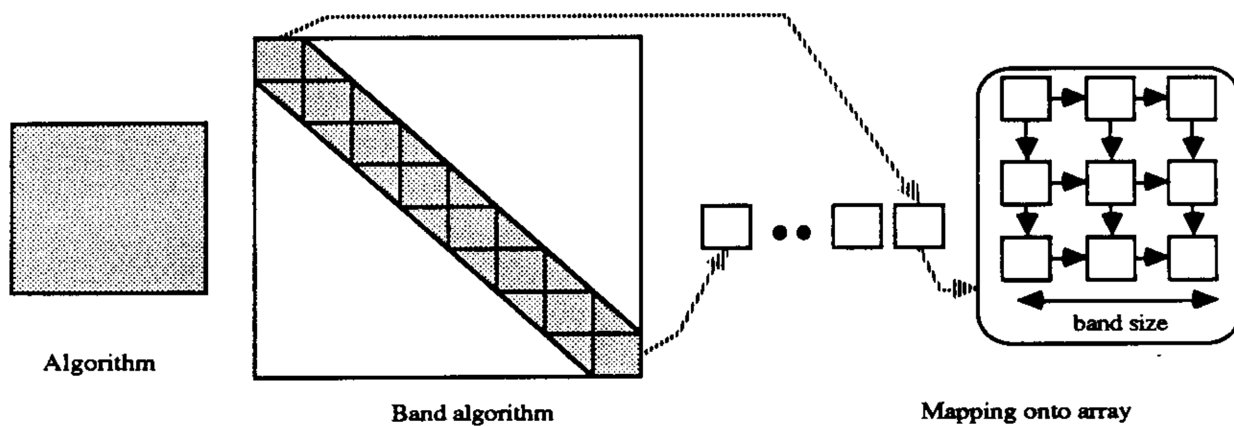


Figure 3: Partitioning an algorithm through decomposition in sub–algorithms

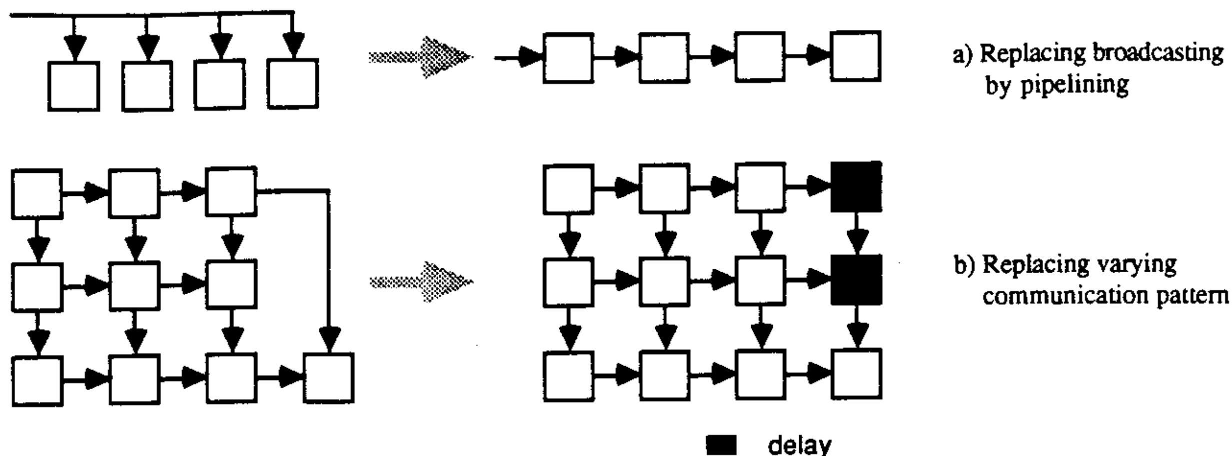


Figure 4: Examples of transformations to remove irregularities in the dependence graph

subalgorithms). We present here a technique based on the dependence graph of algorithms. From the three schemes indicated, we use cut–and–pile because of its generality and smaller memory requirements.

Procedure to partition an algorithm

Our partitioning procedure using the cut–and–pile technique is as follows:

1. Transform the graph to remove properties undesirable for an implementation, such as data broadcasting or bi–directional data flow. Procedures for these purposes have been proposed in [19,20]. Examples are shown in Figure 4. Figure 4a depicts replacement of data broadcasting by pipelined data transfer through the nodes and Figure 4b illustrates a transformation to remove a varying communication pattern among nodes by adding delays.

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, that is, with simple communication requirements. An example is shown in Figure 5, where sets of consecutive nodes in diagonal 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.

   The selection of primitive nodes composing a G–node should fulfill the following requirements:

   (a) It should reduce the G–graph communication requirements, that is produce a G–graph with data dependences among neighbor nodes and simple communication pattern.

   (b) To achieve good utilization of cells in an array, it should attempt to obtain sub–G–graphs where all nodes have the same computation time (i.e., G–nodes composed of the same number of primitive nodes). Sub–G–graphs may correspond to horizontal, vertical, or diagonal paths of the G–graph, or blocks of nodes of the graph.

   (c) The total number of G–nodes should be much larger than the number of cells available in the target array, so that there is enough flexibility in the scheduling process later. Moreover, the structure of the G–graph should be two–dimensional for the same reasons.

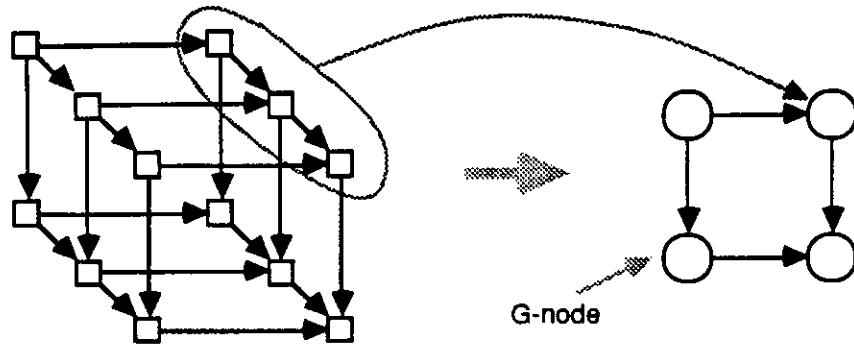


Figure 5: Collapsing primitive nodes into G–nodes

An example of possible alternatives in composing G–nodes is shown in Figure 6, where a dependence graph has been grouped by horizontal, vertical, and diagonal paths. The G–graph resulting in all cases exhibits two–dimensional communication requirements, but patterns of computation time of G–nodes are different. The impact of such patterns in partitioned implementations will be discussed later.

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 7. G–sets scheduled successively are executed in overlapped (pipelined) manner in the array. The selection of G–sets depends on the structure of the target array. In addition, for maximal utilization, all nodes in a G–set should have the same computation time.

In general, the G–graph obtained with our procedure can be directly used to implement an array for a fixed–size problem. However, since the G–graph might be composed of nodes with different computation time, its direct implementation could lead to low utilization of cells.

The approach described above differs from our scheme to design arrays for fixed–size problems [18,19,20], which is also based on dependence graphs, in the following aspects:

- For fixed–size problems, the dependence graph is transformed to incorporate implementation restrictions that include the number of cells available in the array. Consequently, one transformation consists on reducing the number of nodes in the graph so that it matches the interconnection structure and number of cells in the array. For good utilization of the resulting array, such transformation must attempt to *obtain all nodes in the transformed graph with the same computation time.*

  On the other hand, cut–and–pile with good utilization as a goal doesn't require that all G–nodes have the same computation time because not all such nodes will be computed simultaneously. Instead, the objective is to *obtain G–sets whose nodes have the same computation time,* since the nodes of a set will be executed concurrently and different sets will be executed sequentially.

  The differences among these two grouping mechanisms are depicted in Figure 8. For fixed–size problems, the dependence graph is divided into equally sized groups. On the other hand.

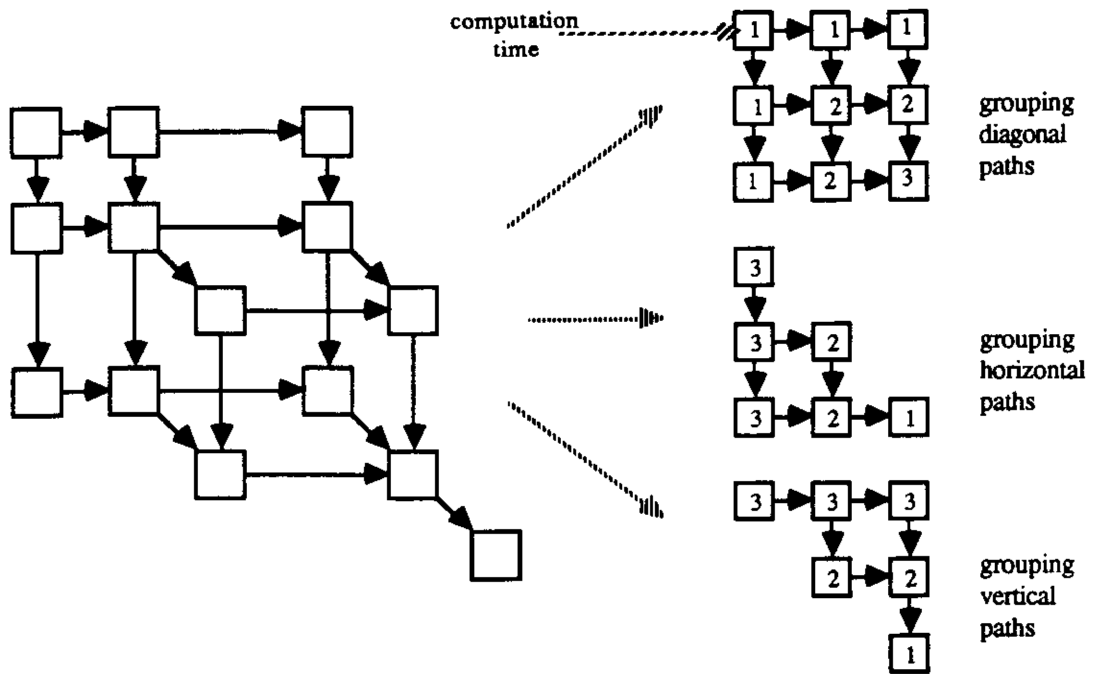


Figure 6: Alternatives in grouping primitive nodes into G–nodes

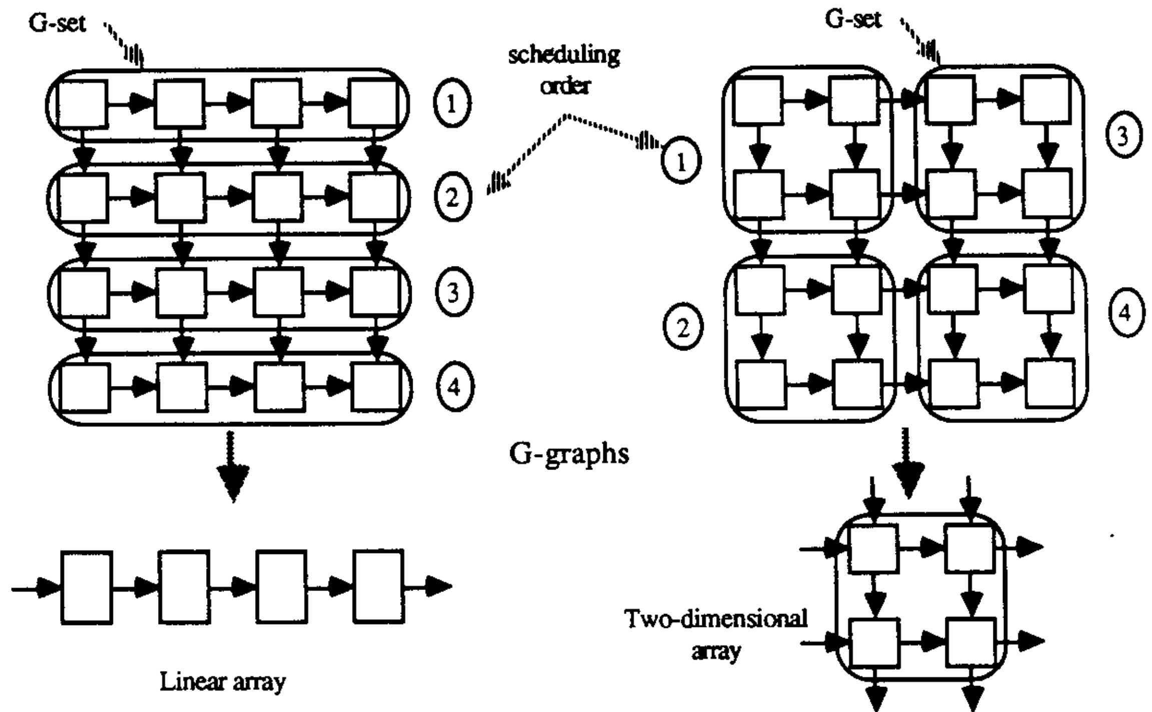


Figure 7: Mapping G–sets onto arrays

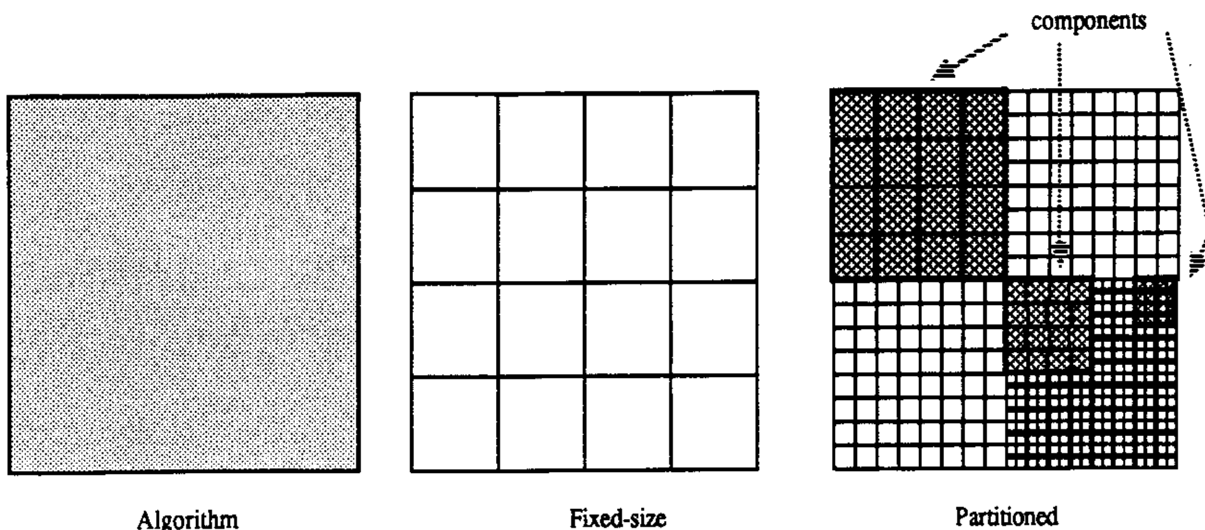


Figure 8: Computation time of G–nodes for fixed–sized vs. partitioned problems

partitioned implementations do not require such an uniform grouping. Instead, partitioning only requires that the nodes of a G–set have the same computation time.

- In fixed–size problems, the fully–parallel graph is transformed into another *graph whose size depends on the size of the problem,* because each of the resulting nodes is mapped onto a cell of the array. In partitioning, the dependence graph is divided in such a way that the *number and complexity of G–nodes can be selected with more freedom,* because the G–graph will be executed in partitioned mode in the array. In particular, for matrices of size n and a target array with m PEs, we have to collapse all nodes of the dependence graph into k sets of m G–nodes each (where $n < km$).

  Figure 9 depicts the differences among these grouping schemes. For fixed–size problems, the dependence graph is divided into as many components as the size of the array. For partitioned cases, the algorithm can be divided into some complex G–nodes, or into many simple G–nodes as shown in the figure.

3 Partitioned Computation of Transitive Closure

We present now the application of the proposed partitioning technique to the design of arrays to compute transitive closure of a directed graph. We first describe briefly the algorithm and then apply the three–step procedure indicated in Section 2.

3.1 The transitive closure problem

A directed graph G is a tuple $G(V, E)$, where V is the set of nodes and E is the set of edges in the graph. G can be described by the *adjacency matrix* A, where element $a_{ij} = 1$ if there is an edge from node i to node j or $i = j$, otherwise $a_{ij} = 0$.

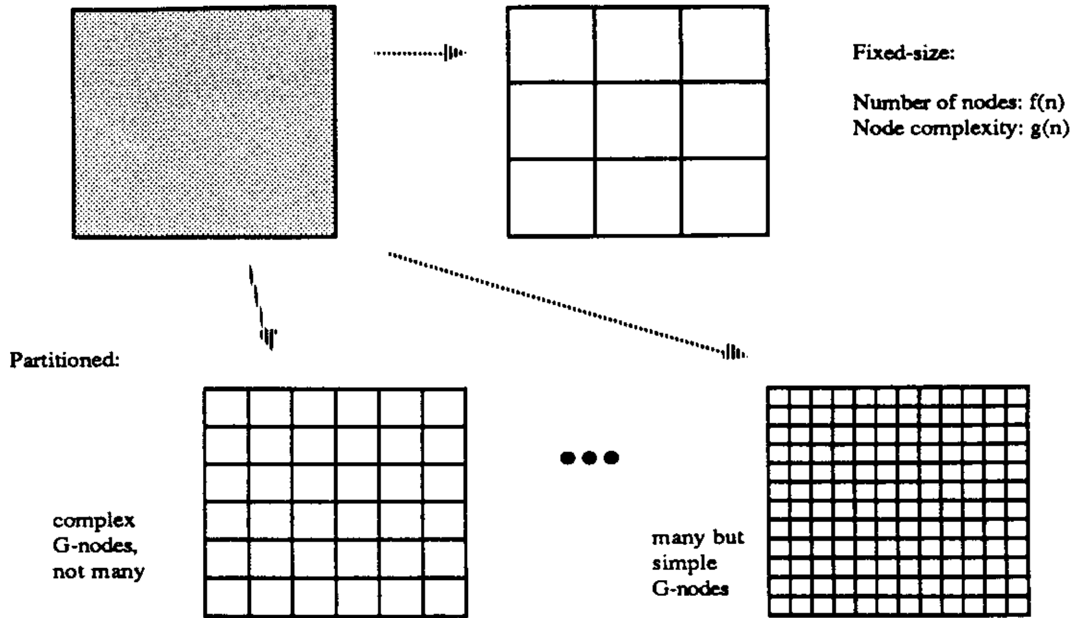


Figure 9: Number of G-nodes for fixed-sized vs. partitioned problems

A directed graph $G^+(V, E^+)$ is called the *transitive closure of G* if it has the same vertex set as G and has an edge from node v to node w if and only if there is a path of length zero or more from v to w in G. G^+ can be described by the *adjacency matrix* A^+.

The computation of the transitive closure of a graph is usually performed by Warshall's algorithm [24]. Given the adjacency matrix A, then A^+ is obtained through the application of the following recurrence:

For k **from** 1 **to** n
For i **from** 1 **to** n
For j **from** 1 **to** n

$$x_{ij}^{k} \leftarrow x_{ij}^{k-1} \oplus (x_{ik}^{k-1} \otimes x_{kj}^{k-1})$$

In this expression, $X^0 = A$, $A^+ = X^n$ and the operators $\oplus$ and $\otimes$ stand for binary-OR and binary-AND, respectively.

The fully-parallel dependence graph of the transitive closure algorithm is shown in Figure 10, for a problem of size $n = 4$ (i.e., to compute the transitive closure of a directed graph with four nodes). The graph has four levels, where each level corresponds to one iteration of the outermost index in the algorithm above. At each level of the graph there is broadcasting of an entire row of the adjacency matrix and also broadcasting of single elements of such matrix, as shown in the figure.

Some evaluations of the expression in the algorithm above do not change the corresponding x_{ij}^k. In particular, the value of a diagonal element in the adjacency matrix is always 1, because a node in the directed graph is always adjacent to itself. In addition, for $k = i$ or $k = j$ one of the

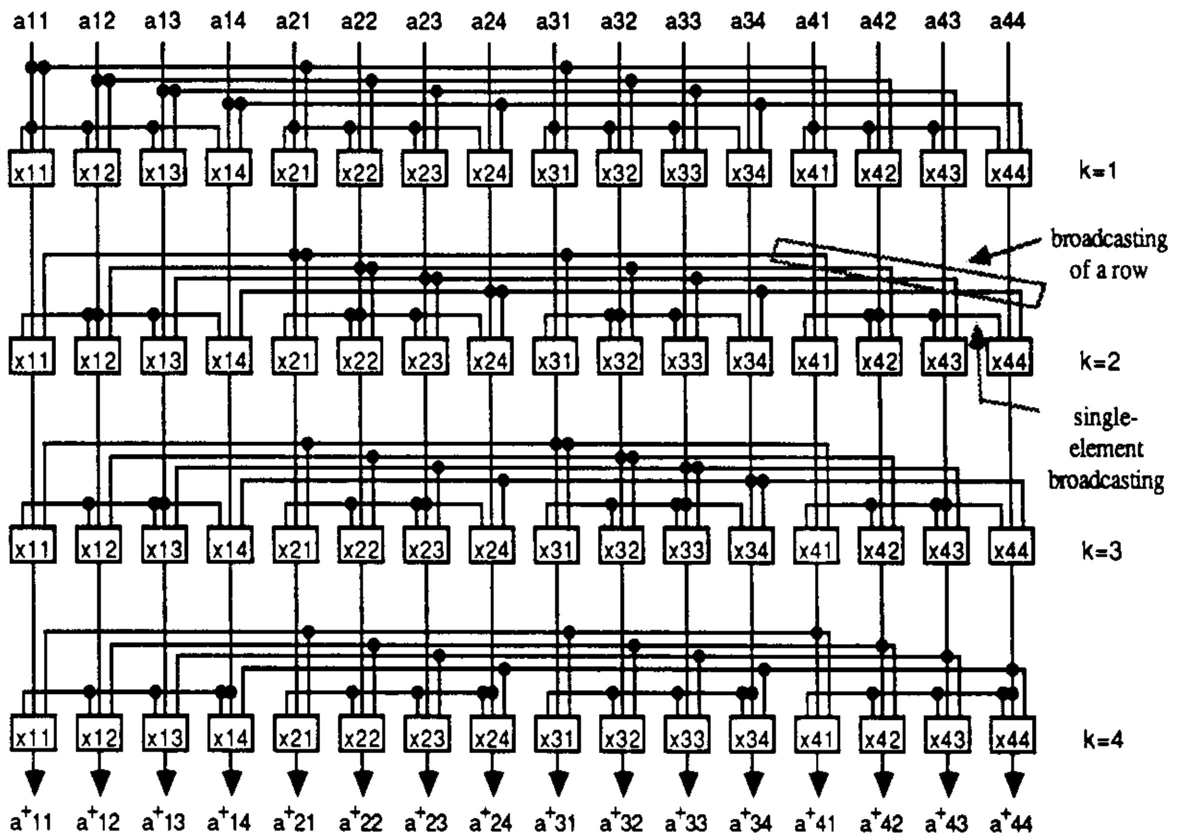


Figure 10: Fully-parallel dependence graph of transitive closure

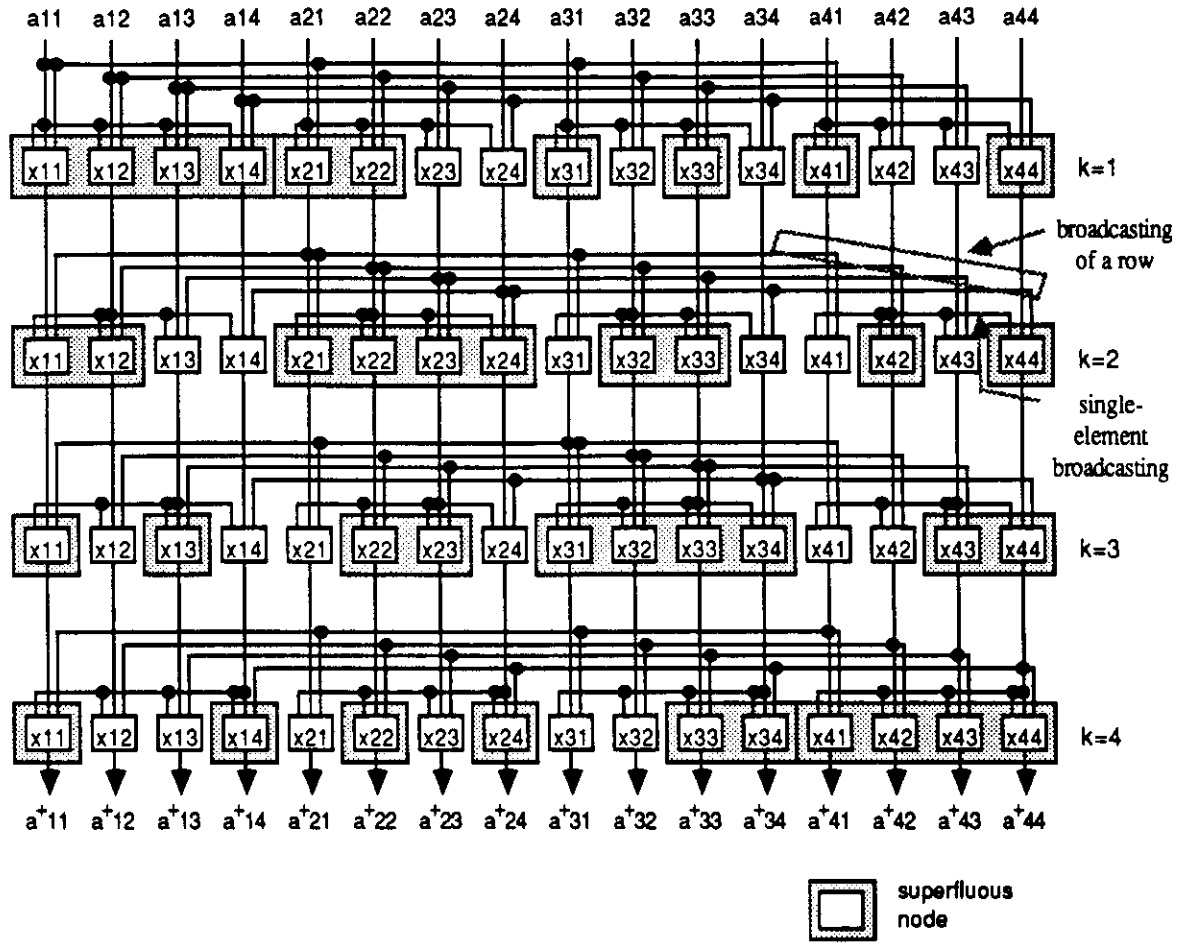


Figure 11: Superfluous nodes in the transitive closure fully–parallel dependence graph

two operands in $x_{ik}^{k-1} \otimes x_{kj}^{k-1}$ becomes x_{kk}^{k-1} which is a diagonal element and thus always equal to 1. Consequently, the result from the $\otimes$ operation is equal to the second operand, the $\oplus$ operator gets two identical operands (x_{ik}^{k-1} or x_{kj}^{k-1}) and the result is that same operand. These properties can be utilized to simplify the design of arrays to compute the transitive closure and to reduce the complexity of the algorithm since fewer operations need to be performed. Figure 11 shows the nodes in the dependence graph which are superfluous (i.e., they do not need to be computed).

3.2 Arrays for partitioned computation of transitive closure

We apply now the procedure proposed in Section 2 to the computation of the transitive closure of a directed graph. The fully–parallel dependence graph shown in Figure 11 is not suitable for implementation, because it exhibits broadcasting of data and complex communication requirements. We address these issues first, according to the procedure described previously.

There are two types of data broadcasting in Figure 11, as indicated before. At the *k-th* level of the graph, data elements from row k of the matrix are broadcasted to all other rows. Moreover,

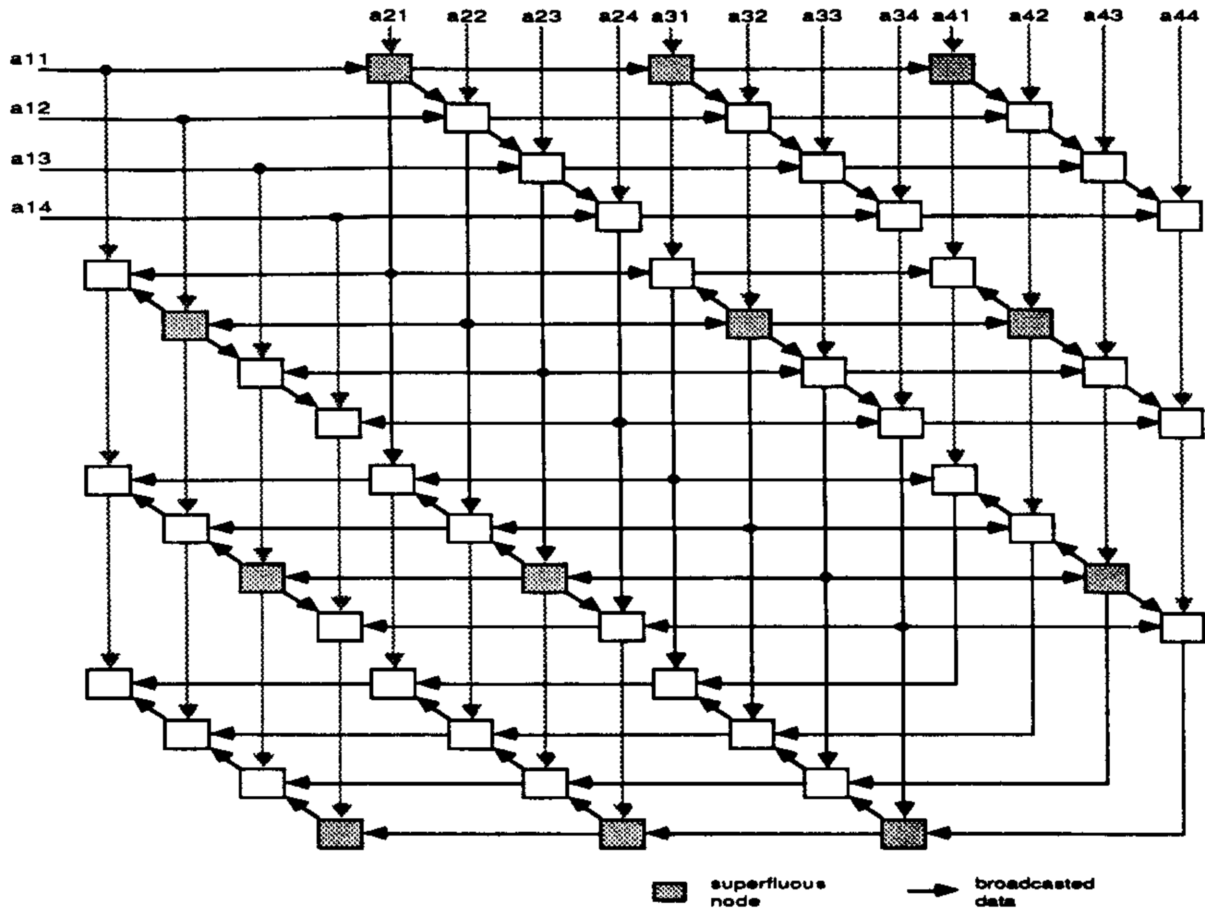


Figure 12: Replacing broadcasting by pipelining in transitive closure dependence graph

the *k-th* element of each row of the matrix is broadcasted to all other elements within each row. Because of the varying pattern of broadcasting, other researchers have considered transitive closure an *irregular* algorithm [23].

We transform the graph replacing broadcasting by pipelining, as suggested in [19]. Given that the *k-th* row of the adjacency matrix remains unchanged at the *k-th* level of the graph, we remove the nodes corresponding to updating those values and draw the flow of data for such row horizontally and intersecting with the flow of data of the other rows of the matrix. Such modification is shown in Figure 12. Data which is evaluated at each level of the graph flows vertically, while the data element broadcasted within each row of the matrix flows diagonally through the row.

Because of the varying source of broadcasting, the transformed graph in Figure 12 exhibits bi-directional flow of data. However, this bi-directional flow can be eliminated by moving nodes dependent on the broadcasted data to one side of the source of broadcasting. We have described such transformation as an approach to solve this problem in dependence graphs [20]. In this case, the transformation is applied in two steps: first, nodes to the left of sources of horizontal broadcasting are flipped to the right end of each row of the graph, as shown in Figure 13a. Then, nodes above sources of diagonal broadcasting are flipped to the bottom end of such diagonals, as shown in Figure 13b. The graph obtained as a result of these two transformations is shown in

Figure 14.

The dependence graph in Figure 14 still exhibits some characteristics not suitable for partitioning. The communication requirements among strips of the graph are not identical for every node (or sets of nodes) within a strip. In particular, a set of diagonal nodes in the inner part of a strip communicates with the set of diagonal nodes to the right on the same strip and with the set below it on the next strip, as shown in Figure 15a. In contrast, the set of diagonal nodes at the rightmost boundary of a strip communicates with two sets on the next strip, while the leftmost set communicates with a set on the next strip not below it but the one shifted to the right, as illustrated in Figure 15b. This varying type of communications complicates the grouping of primitive nodes into G–nodes, because such grouping attempts to reduce the communication requirements of the resulting G–graph. Such varying communication pattern can be removed by adding delay nodes to the part of the graph that is different. These delays are placed with the same communication structure that dominates the graph, as proposed in [20]. The transformation is shown in Figure 15c, and the resulting graph is shown in Figure 16.

Once properties of the original dependence graph not suitable for partitioning have been eliminated, we apply the remaining of our partitioning procedure described in Section 2. We first transform the graph into a G–graph by selecting sets of primitive nodes in such a way as to reduce communication requirements and obtain G–nodes with the same computation time. In this case, diagonal paths are a good alternative for grouping, because nodes in such paths communicate among themselves in a repetitive manner and all paths have the same number of primitive nodes. The result of performing such grouping is the G–graph shown in Figure 17.

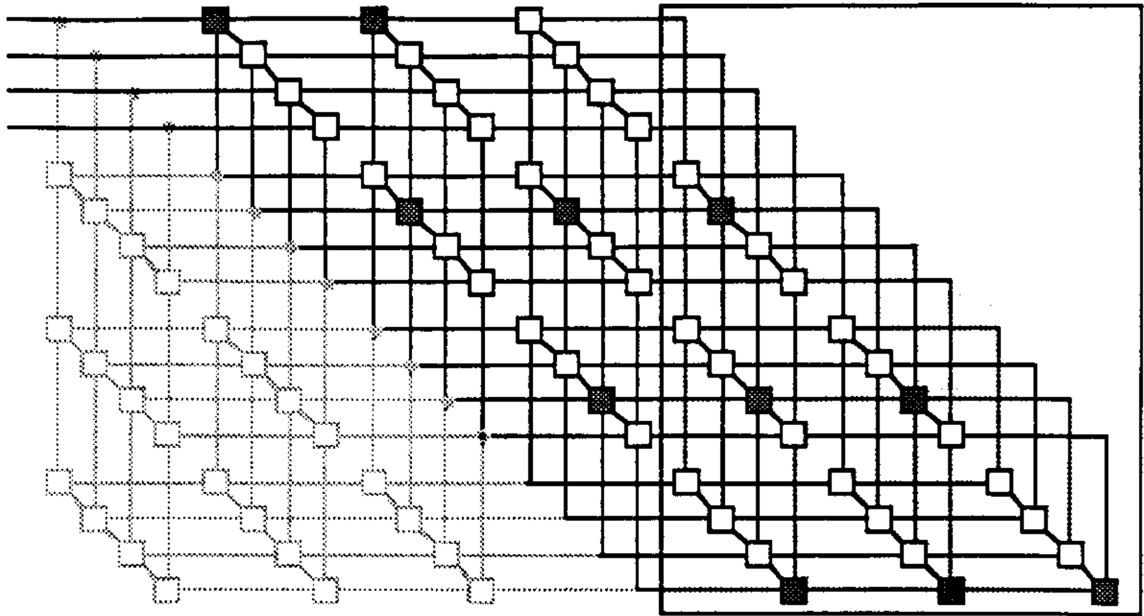
As an aside, Figure 17 is suitable for direct implementation as an array for fixed–size problems. Such an array achieves maximum utilization because all G–nodes have the same computation time and data flows in pipelined manner through the array. Throughput is $1/n$ because the computation time of G–nodes is n cycles. Successive instances of the algorithm can be chained without restrictions. This array is simpler than the one proposed in [23] because it has a single communication path between cells and no control complexity. Furthermore, data transfers and computations are overlapped while the array proposed in [23] requires that "data be first loaded in the nodes and then reused for a period of n cycles" so that "certain control is required in the systolic array."

Another advantage of the array for fixed–size problems described above relies on the simplicity of its derivation through the graph–based methodology. This is in contrast with the scheme in [23], which uses a rather complex mathematical approach. Furthermore, the G–graph in Figure 17 can be collapsed into a linear structure by grouping each horizontal path into single nodes. The resulting graph can be directly mapped onto a linear array with throughput $[n(n+1)]^{-1}$ and all cells fully utilized.

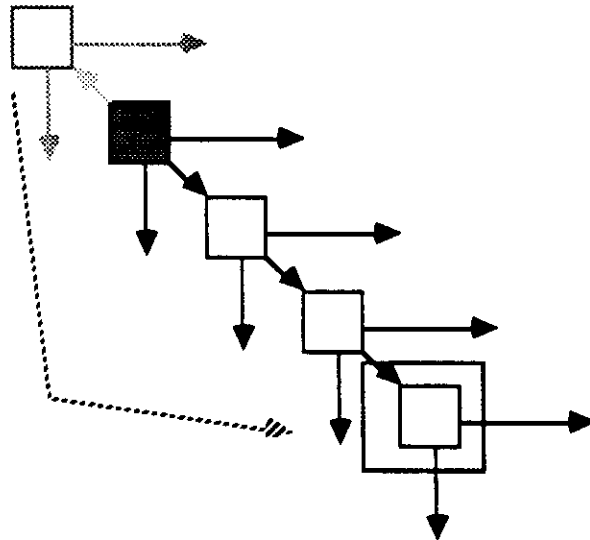
Arrays to compute the transitive closure in partitioned mode can be derived directly from the G–graph in Figure 17, as we describe next.

Linear array

Let's assume that we want to partition the computation of transitive closure of a directed graph with n nodes so that it fits in a linear structure with m cells, where $m \ll n$. We map G–sets from the transformed graph onto a linear array by selecting G–sets of m G–nodes from horizontal paths.



(a) Removing horizontal bidirectional broadcasting



(b) Removing diagonal bidirectional broadcasting

Figure 13: Removing bi-directional broadcasting in dependence graph of transitive closure

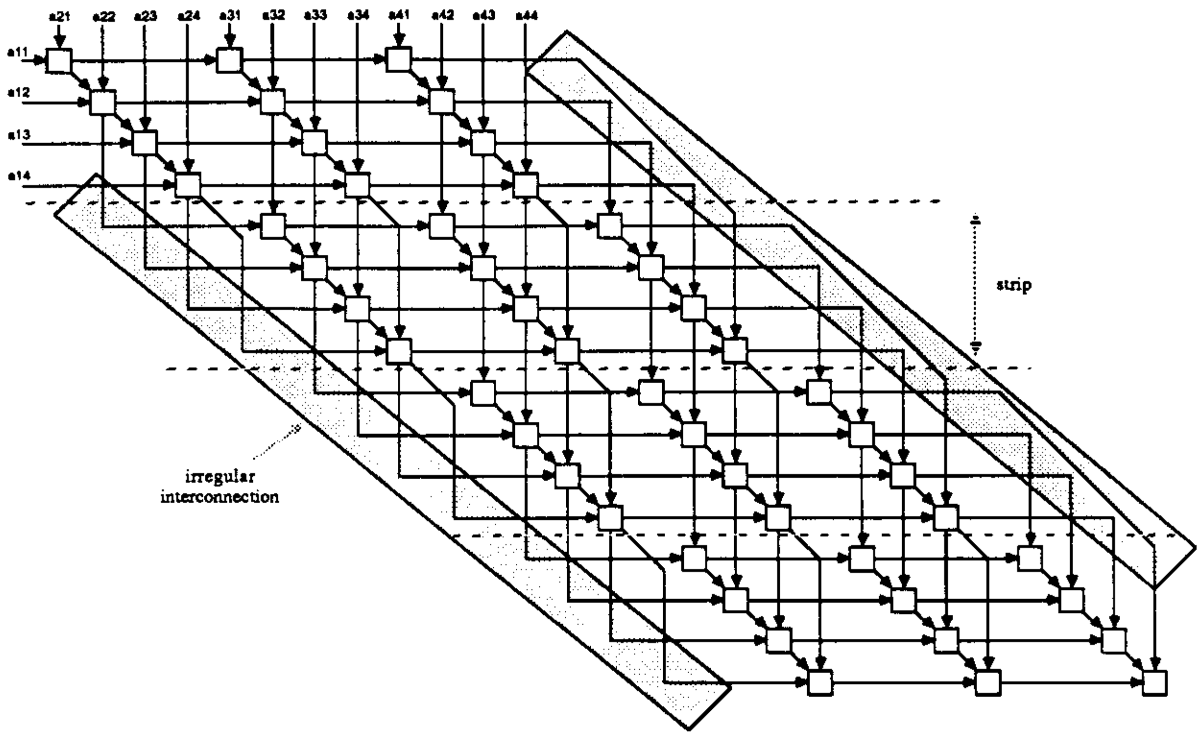


Figure 14: Transitive closure dependence graph with uni-directional broadcasting

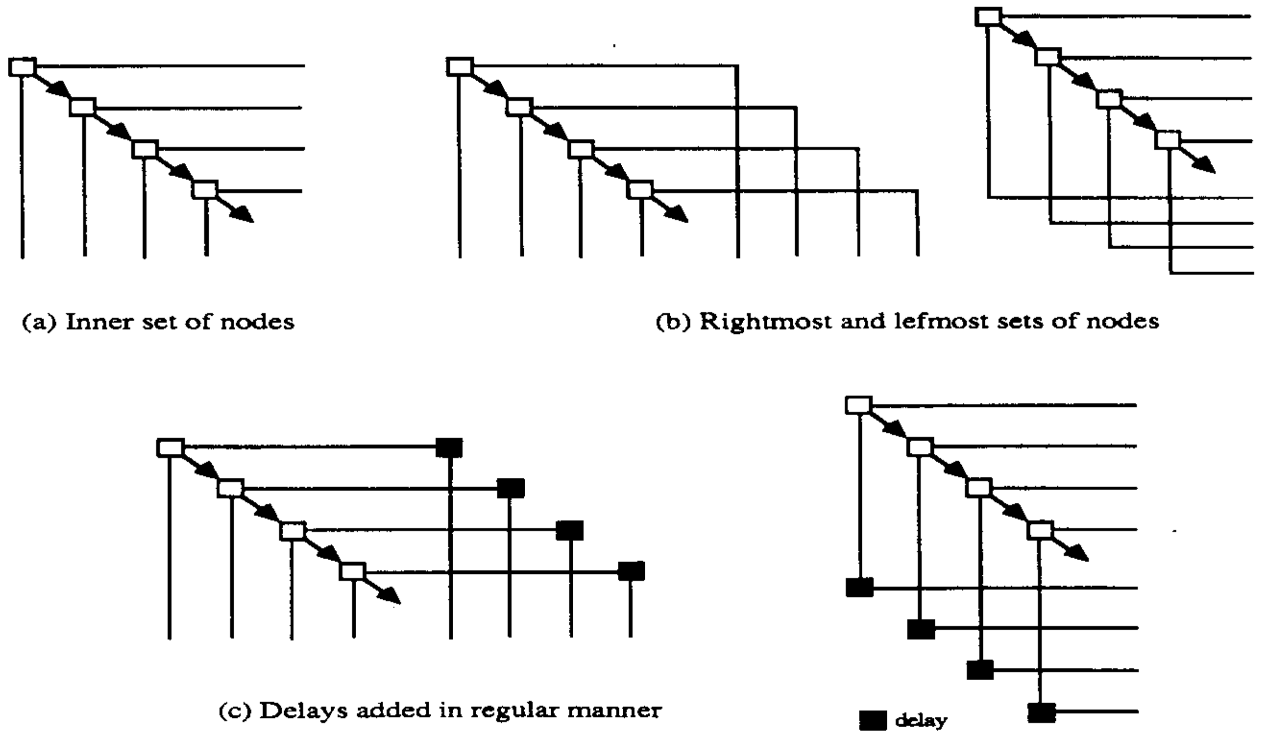


Figure 15: Removing irregular communications in transitive closure dependence graph

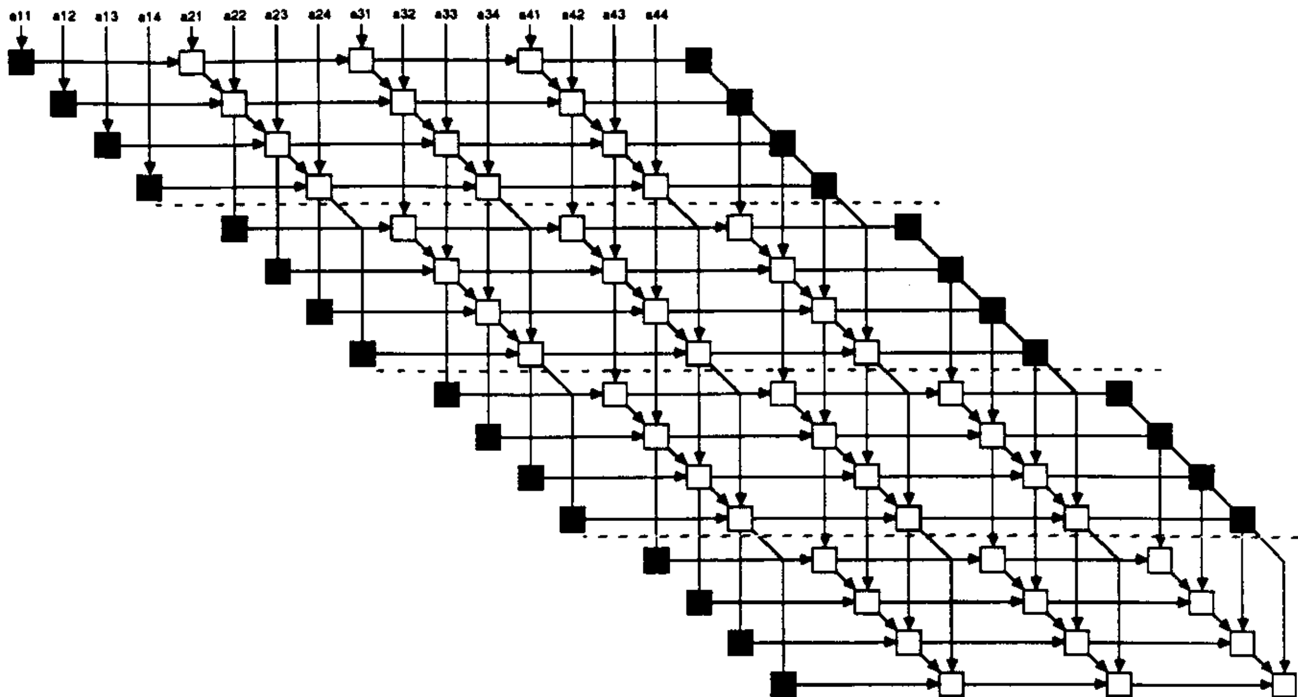


Figure 16: Regular transitive closure dependence graph

as shown in Figure 18. Scheduling of such G–sets is discussed later. Intermediate results from G–sets are saved in external memories. Such data is available at the boundary of the set, so that saving it in external memories is straight–forward.

The structure resulting from this approach, shown in Figure 18, enjoys maximal utilization because all G–nodes executed concurrently have the same computation time, except when executing boundary sets in some horizontal paths that might not use all cells in the array. The number of connections to external memories is $m + 1$.

Two–dimensional array

We map now the computation of transitive closure in partitioned mode onto a two–dimensional array. Mapping the transformed dependence 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 selected as square blocks of $\sqrt{m}$ by $\sqrt{m}$ nodes, excepting sets at the boundaries of the G–graph which are composed of triangular blocks of G–nodes, as shown in Figure 19a. As in the linear case, intermediate results are saved in external memories. The structure resulting from this approach is shown in Figure 19b. Utilization of this array is maximal, except when executing boundary sets because such sets do not use all cells in the array. The number of connections to external memories is $2\sqrt{m}$.

To use the arrays obtained above it is necessary to schedule the execution of G–sets. We discuss

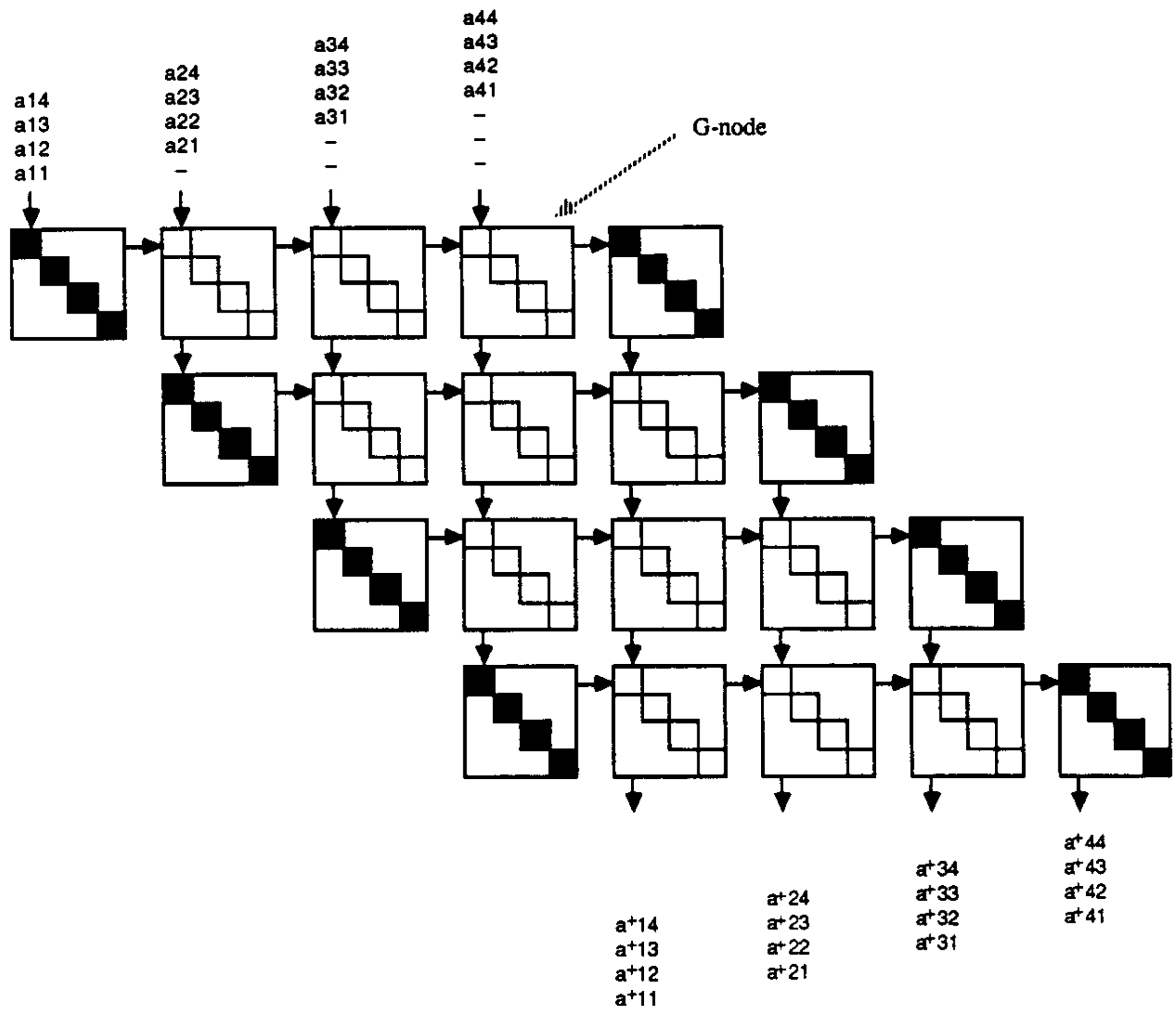


Figure 17: Transitive closure G–graph

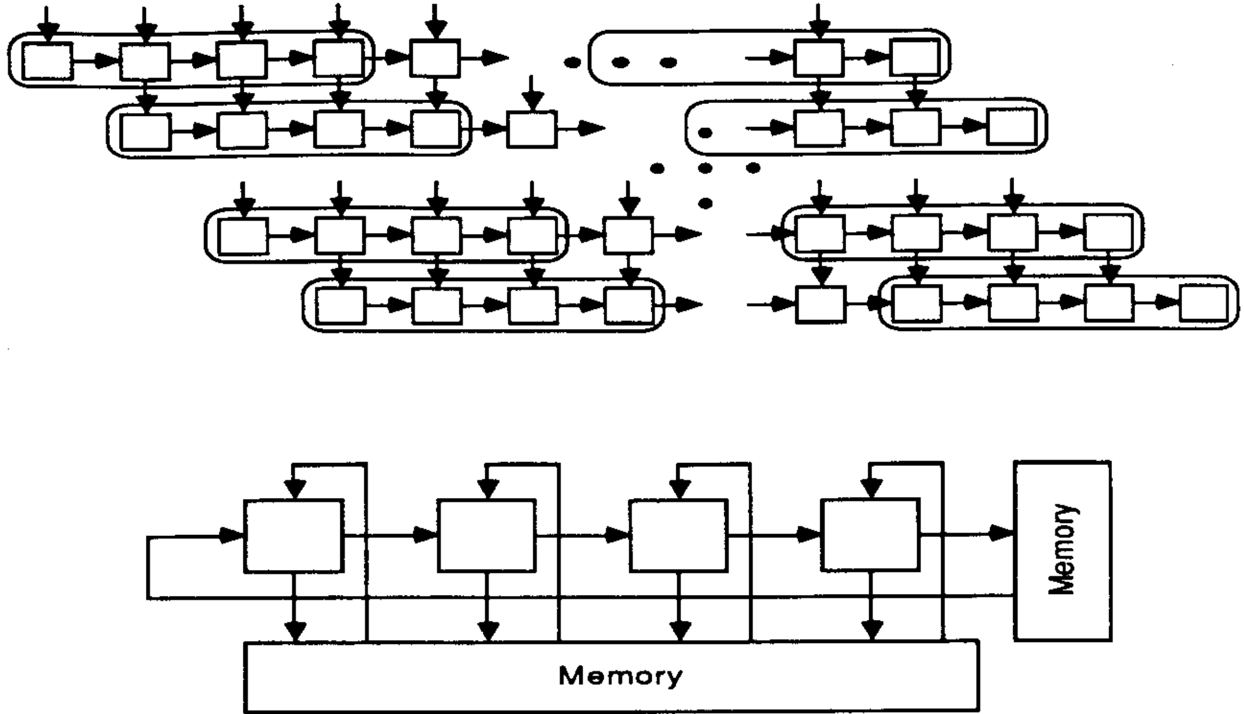


Figure 18: Mapping transitive closure G–graph onto a linear array

now such scheduling for both arrays. We also discuss the I/O bandwidth requirements in each case.

Scheduling and I/O requirements in partitioned transitive closure

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 20 (this graph can be regarded as the internal portion of a large G–graph). G–nodes in horizontal paths have identical computation time and data flows in pipelined manner. Nodes in Figure 20 have been tagged with their earliest scheduling time relative to a reference time t_i.

Scheduling of G–sets must take into account the dependences among G–sets. Because of the pipelined nature of data flow within the array, a G–set can't be scheduled for computation until the required data produced by 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 m cells. Since $m \ll n$, data needed to schedule execution of the next 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 paths. Scheduling of G–sets can be done by horizontal or vertical paths. For I/O bandwidth reasons discussed below, we choose to schedule G–sets by vertical paths as depicted in Figure 20a. 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, as illustrated in

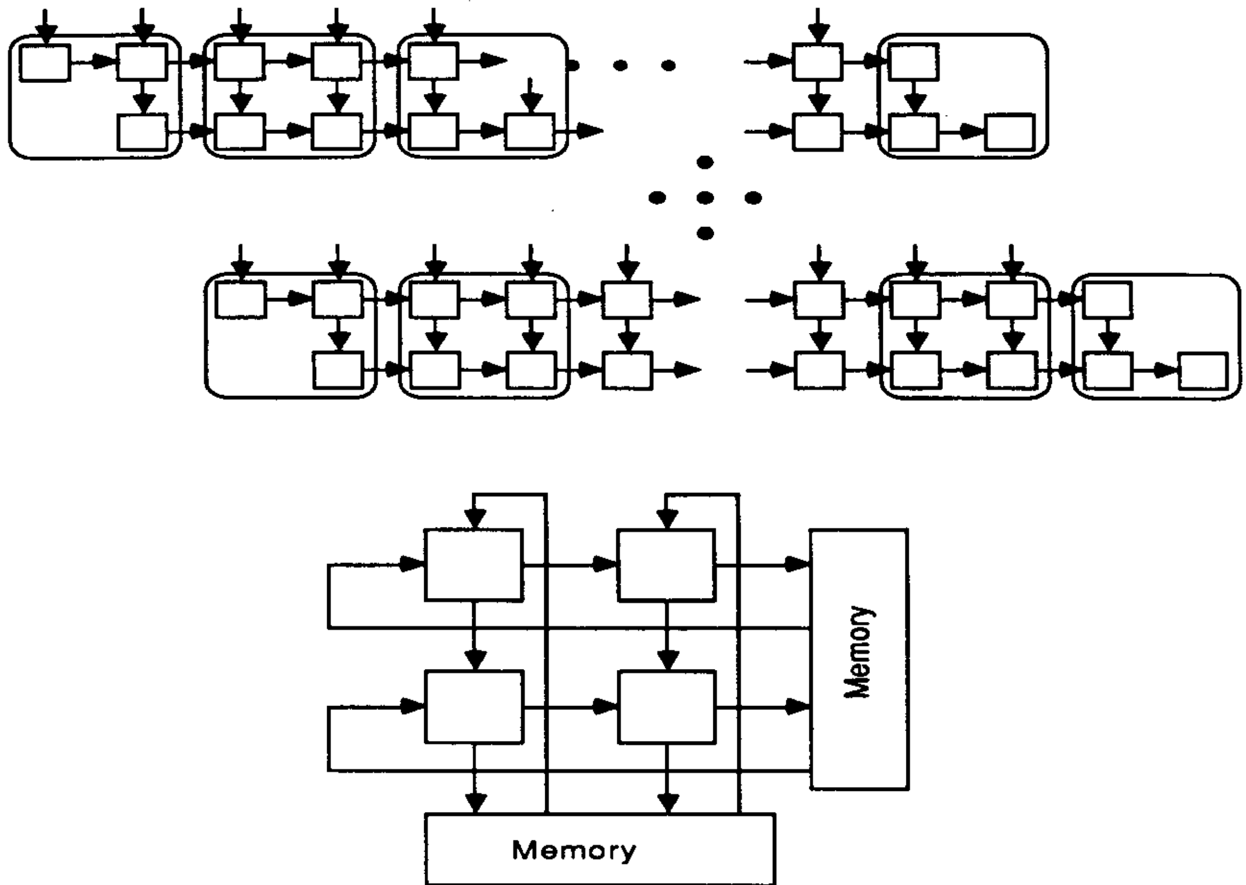
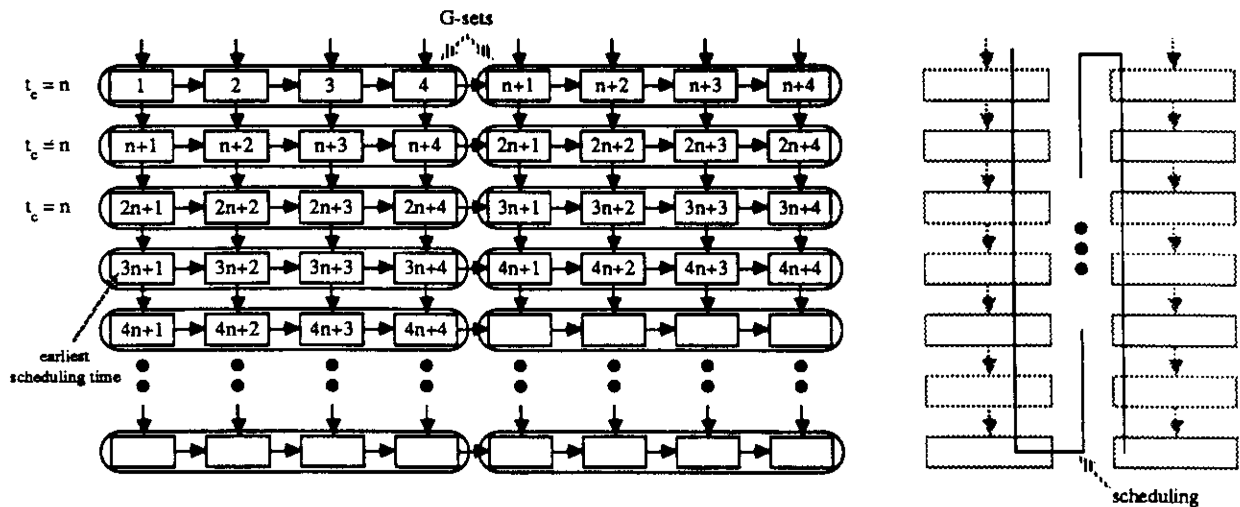
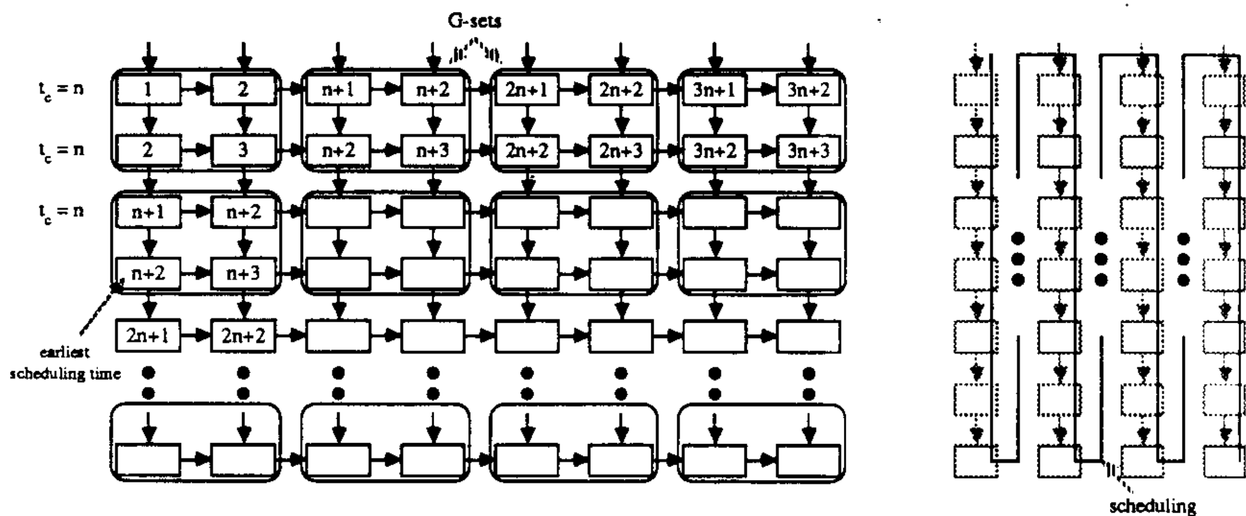


Figure 19: Mapping transitive closure G–graph onto two–dimensional array



(a) - Scheduling G-graph into linear array



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

Figure 20: Scheduling G–graph into linear and two–dimensional array

Figure 20b.

A host feeding input data to the array needs to provide only the elements appearing as input to nodes at the topmost horizontal path of the G–graph. Intermediate values are saved in and obtained from external memories attached to the array. To reduce the rate at which data has to be provided to the array, nodes at the top of the G–graph shouldn't be scheduled consecutively. In such a case, the host needs to feed data to the array at a rate lower than one input per cell per cycle and utilization of I/O connections can be increased by decoupling computation from data transfer, as proposed in [18,19]. Such approach leads to the I/O structure shown in Figure 21, 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 memory. 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 21 have been tagged with their scheduling time, as determined above. Since each node at the top of the G–graph receives n data elements from the host, I/O bandwidth is given by

$$D_{I/O} = \frac{nm}{\sum_{k=1}^{n} t_{c_k}} = \frac{nm}{n^2} = \frac{m}{n}$$

where t_{c_k} is the 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 Evaluating the performance of arrays for partitioned execution

In this section, we discuss the evaluation of arrays for partitioned execution of algorithms. First we introduce the performance measures used and later apply such measures to the arrays derived above for the transitive closure algorithm.

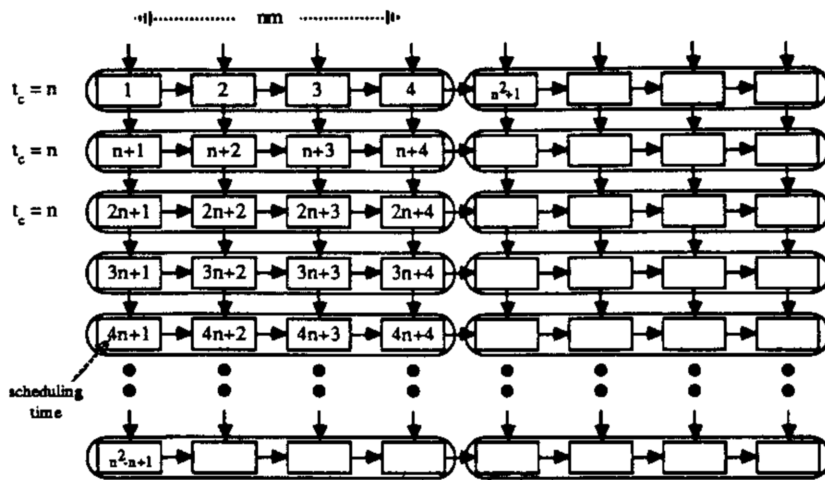
4.1 Evaluation measures

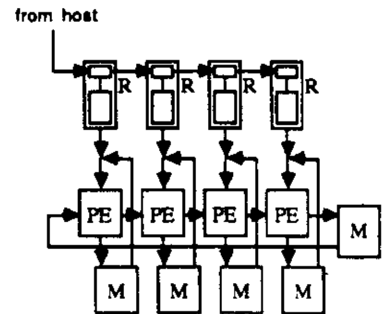
To determine the performance of arrays for partitioned execution, we fix the number of cells available for an implementation to m and use the following measures:

- Throughput T
- Utilization U
- Input/output bandwidth $D_{I/O}$
- Overhead due to partitioning

These parameters are computed with information obtained from the dependence graphs, both the original and the transformed graphs.

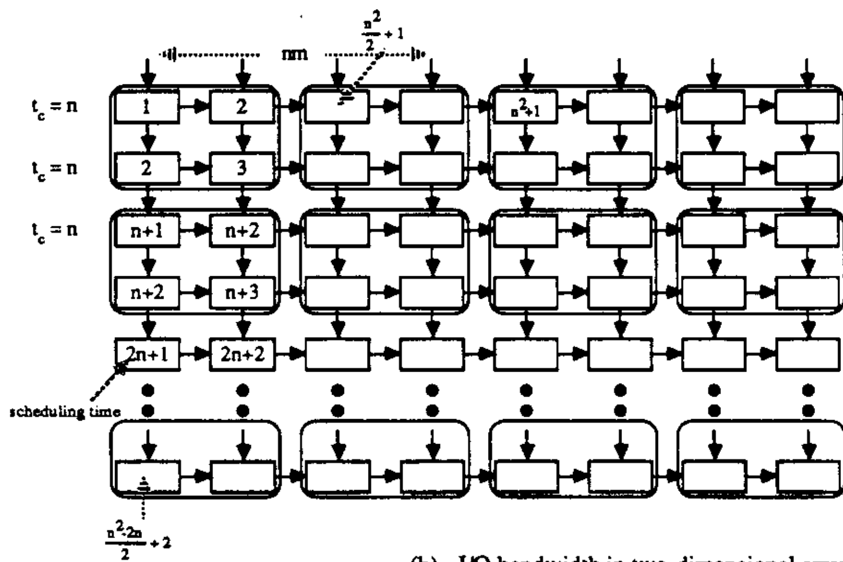
Throughput is given by $T^{-1} = \sum_i(\tau_i^{-1} + d_i)$, where τ_i is the throughput of the *i–th* G–set mapped onto the array and d_i is the overhead (if any) due to partitioning. In turn, $\tau_i = t_l^{-1}$, where t_l is the longest computation time of a node in the *i–th* G–set.

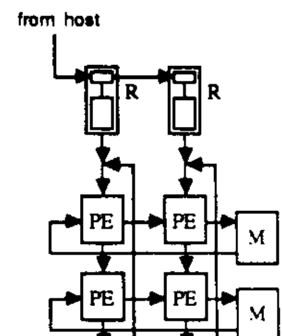




I/O BW = m/n

(a) -I/O bandwidth in linear array





I/O BW = m/n

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

Figure 21: I/O bandwidth in partitioned transitive closure

Overhead in partitioned execution arises when the implementation needs to execute actions which are not part of the algorithm. In particular, this overhead refers to time spent in data transfers such as loading and unloading values into/from the array. If flow of data through the G–nodes is pipelined, then $d_i = 0$ because no such overhead exists.

Utilization of the arrays is computed as $U = (\sum_i n_i t_i)/(m/T)$, where n_i is the number of nodes in the fully–parallel dependence graph with computation time t_i. The expression $\sum_i n_i t_i$ corresponds to the total number of nodes in the original dependence graph. Computing such number can be simplified by taking advantage of properties of the dependence graph, so that computing utilization requires to obtain the throughput of the implementation as described above and compute the total number of nodes in the graph.

Consequently, all the information required to evaluate the performance of partitioned execution of an algorithm is readily available from the dependence graphs used to derive the implementation.

4.2 Evaluation of arrays for partitioned computation of transitive closure

We now evaluate the performance of linear and two–dimensional arrays for partitioned computation of transitive closure derived earlier, using the performance measures indicated above.

The number of nodes in the fully–parallel dependence graph shown in Figure 11 is $N = n^3$. However, as we stated earlier, the computation of several of these nodes is superfluous because the value of the computed element doesn't change in such nodes. The actual number of nodes that must be computed is $N = n(n-1)(n-2) = n^3 - 3n^2 + 2n$, because there are n levels in the graph each with $(n-1)$ rows of the adjacency matrix that need to be evaluated. In each row $(n-2)$ elements are computed, since the computation of the diagonal element and the element broadcasted within each row are superfluous.

Linear array

Mapping the G–graph onto a linear array with m cells is done with $n(n+1)/m$ G–sets, because the graph has n horizontal paths and each of such paths is mapped with $(n+1)/m$ G–sets. Each node in a G–set has computation time $t_c = n$. Consequently, the linear array exhibits throughput

$$T = \left[\sum_i t_l\right]^{-1} = \left[\frac{n(n+1)}{m} n\right]^{-1} = \frac{m}{n^2(n+1)}$$

and utilization of such an array is

$$U = \frac{N}{m/T} = \frac{n(n-1)(n-2)}{mn^2(n+1)/m} = \frac{(n-1)(n-2)}{n(n+1)} \rightarrow 1$$

The computation of the algorithm in partitioned mode exhibits no overhead.

Two–dimensional array

Mapping the G–graph onto a two–dimensional array is done with $(n/\sqrt{m})((n+1)/\sqrt{m}) = n(n+1)/m$ G–sets. Therefore, the two–dimensional array has the same throughput and utilization as

the linear array with the same number of cells. The computation of the algorithm in partitioned mode exhibits no overhead.

From the analysis above, we conclude that the linear scheme is more convenient than the two-dimensional one, because it has a simpler structure and exhibits the same throughput, utilization and I/O bandwidth than the two-dimensional array. This analysis is valid not only for transitive closure but also for other algorithms, because for large-size data any G-graph will have a large *internal portion* such as the one shown in Figure 20. This is a consequence of the methodology, because the transformations performed on the graph have as objectives to reduce communication requirements of the G-graph and to obtain a two-dimensional G-graph. Such internal portion will account for most of the computation time of an algorithm.

The derivation of the arrays above has been facilitated by the structure of the G-graph, in particular by the identical computation time for all G-nodes. However, there are cases where some property of the G-graph renders the design of arrays more complex than the example considered here. An example of such problems is G-nodes with different computation time. We discuss this issue and its impact on partitioning.

4.3 G-nodes with different computation time

There are cases when it is not possible to obtain all nodes in a G-graph with the same computation time without compromising the communication requirements of the graph. Given the regularity of matrix algorithms, such cases normally lead to graphs where computation time of G-nodes varies in a specific manner across the graph (i.e., monotonically decreasing/increasing computation time across horizontal or vertical paths of a G-graph). Examples are algorithms for LU-decomposition, inverse of non-singular upper triangular matrix, triangularization by Givens rotations, Faddeev algorithm, among others. In such cases, mapping the G-graph for execution in a two-dimensional array implies that it is not possible to select G-sets with the same communication requirements as the structure of the array and all G-nodes with the same computation time. This situation is depicted in Figure 22a, where nodes have been tagged with their computation time. Nodes in vertical paths of this graph have the same computation time, but successive vertical paths have decreasing computation time. Any two-dimensional G-set will include nodes with different computation time as shown in the figure, implying that the utilization of a two-dimensional array will not be maximal.

On the other hand, there usually are vertical or horizontal paths of the G-graph where nodes have identical computation time. In such a case, it is possible to map such G-graph into a linear array and achieve maximal utilization as shown in Figure 22b. Consequently, a linear array for partitioned execution will be more efficient than a two-dimensional one.

5 Conclusions

We have proposed a technique to partition algorithms for execution in arrays, based on dependence graphs of algorithms. This is a transformational approach, where the dependence graph is first transformed to remove properties not suitable for implementation, then converted into a graph with simple communication requirements, and finally mapped onto the target array. This process

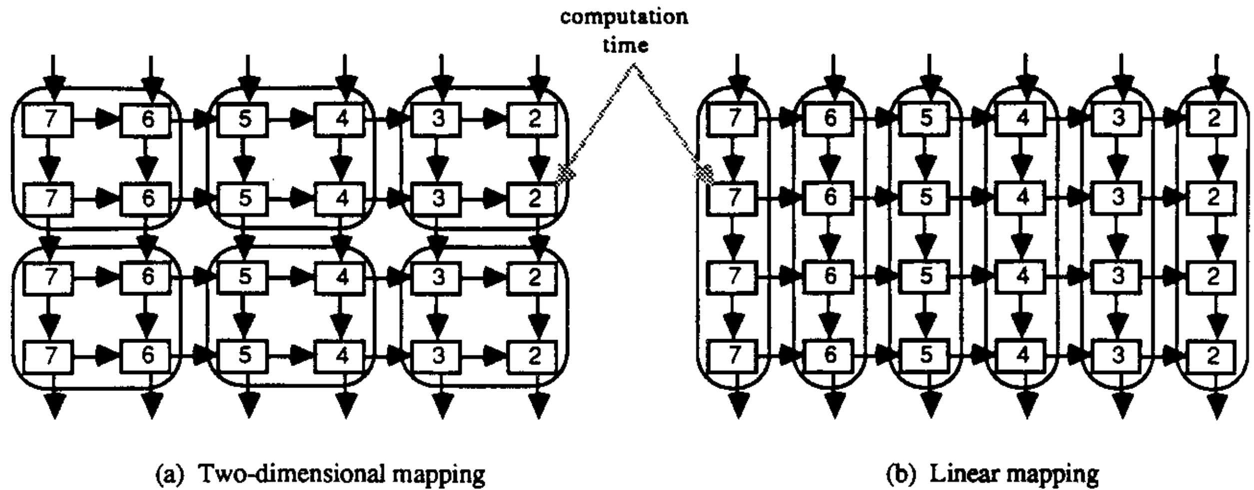


Figure 22: G–nodes with different computation time

is guided by issues such as utilization of PEs, throughput, and I/O bandwidth.

We described the application of such technique to the computation of transitive closure of a directed graph. Through this example, we have shown that the approach is general and powerful. This technique doesn't place restrictions on the algorithms, produces implementations with maximal utilization of cells and no overhead due to partitioning, and allows evaluating trade–offs between linear and two–dimensional structures. Moreover, this graph–based approach is simpler to use than schemes based on mathematical expressions.

We derived linear and two–dimensional arrays for partitioned computation of transitive closure. In the process, we have obtained a dependence graph which is suitable for implementation of a fixed–size array for transitive closure, with better characteristics than structures previously proposed for this algorithm.

We have described important issues in partitioning algorithms, in particular trade–offs between linear and two–dimensional structures. We have shown that, with the same number of cells, linear arrays are simpler, have the same throughput and I/O bandwidth from the host than two–dimensional ones, and might exhibit better utilization. Moreover, linear arrays are more advantageous than two–dimensional ones because they are better suited to incorporate fault–tolerant capabilities. Consequently, we conclude that *linear arrays offer better performance and implementation than two–dimensional arrays for partitioned execution of algorithms.*

References

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

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] S. Kung, *VLSI Array Processors*, pp. 374–382. Prentice Hall, 1988.

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

[11] 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.

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

[13] P. Liu and T. Young, "VLSI array design under constraint of limited I/O bandwidth," *IEEE Transactions on Computers*, vol. C-32, pp. 1160–1170, Dec. 1983.

[14] C. Raghavendra, V. Prasanna-Kumar, and A. Varma, "On systolic processing with bounded I/O bandwidth," in *International Conference on Computer Design*, pp. 632–635, 1985.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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, Feb 1988.

[21] J. Moreno and T. Lang, "On partitioning the Faddeev algorithm," to be published in *International Conference on Systolic Arrays*, 1988.

[22] F. Núñez and N. Torralba, "Transitive closure partitioning and its mapping to a systolic array," in *International Conference on Parallel Processing*, pp. 564–566, 1987.

[23] S. Kung, *VLSI Array Processors*, pp. 248–266. Prentice Hall, 1988.

[24] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.