REDUCING THE NUMBER OF CELLS IN ARRAYS FOR MATRIX COMPUTATIONS

Jaime H. Moreno
Tomas Lang

March 1988
CSD-880014

Reducing the Number of Cells in Arrays for Matrix Computations*

Jaime H. Moreno
3680 Boelter Hall
(213)-825-2266

Tomás Lang
3732E Boelter Hall
(213)-825-6835

Computer Science Department
School of Engineering and Applied Science
University of California
Los Angeles, Calif. 90024-1596

This report is a revised version of Technical Report CSD–870040 "Removing algorithm irregularities in the design of arrays for matrix computations," Computer Science Department, University of California Los Angeles, August 1987.

*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"

Reducing the Number of Cells in Arrays for Matrix Computations

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

Abstract

We propose a procedure to reduce the number of cells in arrays of processing elements (PEs) for matrix computations, as part of a design methodology. Such procedure achieves its objective by reducing the number of nodes in a fully-parallel dependence graph used to describe the algorithm. In addition, we identify some irregularities which complicate the procedure and propose mechanisms to deal with them. The irregularities considered are varying paths length, bi-directional broadcasting and non-regular interconnection pattern between nodes of the dependence graph. The mechanisms include an extension to the procedure proposed for reducing nodes, to cope with varying path length, and transformations to the dependence graph to remove the remaining irregularities considered. The graphs obtained from these mechanisms are suitable for further transformations aimed towards regular graphs or for direct implementation as arrays of PEs. We use LU-decomposition, without and with pivoting, as examples of application of these mechanisms. The methodology produces triangular arrays for this computation, with better utilization than the square arrays formerly proposed for it.

1 Introduction

Matrix computations are the basis for many applications in science and engineering. Examples exist in image and signal processing, pattern recognition, control systems, among others. The evolution in VLSI technology is making possible the cost-effective implementation of many matrix algorithms as a collection of regularly connected processing elements (PEs).

An important problem in the design of arrays of PEs for a given algorithm is the methodology used to derive the structure and interconnection of those arrays. Standard structures (systolic arrays [1]) have been used for these implementations [2]-[10]. Some transformational methodologies have been proposed [11], where a high-level specification of a problem is transformed into a form better suited for implementation. Although the proposed approaches can be useful to accomplish certain design tasks, they either restrict the form of the algorithm (i.e., a recurrence equation) or are unable to incorporate implementation restrictions as part of the methodology.

We have proposed a graph–oriented design methodology for arrays of PEs, with the capability to handle and relate features of the algorithm and the implementation in a unified manner [12]. This is a transformational methodology, which uses a fully–parallel dependence graph as the description of the algorithm. In such a graph, nodes represent the operations and edges correspond to data communications. We assume that all nodes have the same computation time. These graphs are characterized by having all inputs and outputs available in parallel and no loops (i.e., loops are unfolded). Starting from a fully–parallel graph, we apply transformations to incorporate issues such as data broadcasting, data synchronization, interconnection structure, I/O bandwidth, number of PEs, throughput, delay, and utilization of PEs. Preliminary results on the application of this methodology have been reported in [13].

We use the fully–parallel graph to describe an algorithm, because such graph is unique for a given algorithm and exhibits the intrinsic properties of the algorithm. The graph can be used directly to derive an implementation by assigning each node of the graph 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 node with the longest computation time), but may require complex interconnection structure, high I/O bandwidth, and large number of units. Our methodology deals with these problems, while still attempting to preserve the delay and throughput inherent in the dependence graph.

In this paper, we propose a procedure to reduce the number of nodes in the fully–parallel dependence graph with the purpose of reducing the number of cells needed in an implementation. In addition, we identify some irregularities which complicate the procedure, and propose mechanisms to deal with them. These irregularities are varying paths length, bi–directional broadcasting and non–regular interconnection pattern between nodes of the dependence graph. Examples of algorithms exhibiting such irregularities are square of a matrix, LU–decomposition without and with pivoting, matrix triangularization, inverse of non–singular triangular matrix, among others. We extend the procedure proposed for reducing the number of nodes to take into account the varying paths length, and propose transformations to the dependence graph to reduce the other irregularities considered. The graphs obtained from these mechanisms are suitable for further transformations aimed towards regular graphs or for direct implementation as arrays of PEs. We use LU–decomposition, without and with pivoting, as examples of application of our mechanisms.

This work is part of the development of our design methodology. Additional assumptions and details about the methodology are described more fully in [12] and [13].

2 Reducing the Number of Nodes in the Dependence Graph

The fully–parallel graph of a matrix algorithm of the size used in real applications always has too many nodes, so that a pipelined implementation of such graph requires too many units. Transformations to incorporate certain implementations restrictions, such as removing data broadcasting or restricting the input/output bandwidth, may reduce that number. However, in most cases it is necessary to reduce the number of nodes (i.e., reduce the concurrency) further. We achieve this objective by grouping sets of nodes into single nodes. Evidently, the computation time of the new nodes is longer and the throughput of a pipelined implementation based on the transformed graph is lower. (This reduction is complicated by irregularities in the algorithm, as discussed later.) We present now a procedure to perform such reduction.

Reducing the number of nodes in a graph requires a criterion to group set of nodes into single nodes. To achieve a suitable implementation, the grouping should satisfy the following constraints:

- Obtain new nodes with identical computation time, so that the utilization of an array is maximized.
- Obtain a regular interconnection pattern among the new nodes.
- Preserve the length of the critical path in the graph.

It is not always possible to fulfill all these constraints simultaneously. However, such constraints must guide the process of grouping nodes. While doing this grouping, we have to consider the following issues:

i) Grouping nodes of a sub–graph reduces the interconnection requirements and the parallelism. The dependences between the nodes in a group are transformed into dependences within the new node, so that no communication path is required for those nodes. An example of this situation is shown in Figure 1a.

ii) Grouping nodes at different levels in the graph (i.e., nodes which are dependent) reduces the degree of pipelining possible in the graph (i.e., fewer stages) but preserves the delay of the computation. This is in contrast to grouping nodes at the same level, because in such a case the parallelism is reduced and the path length is increased, possibly increasing the length of the critical path in the graph. Grouping nodes at different levels is shown in Figure 1b.

iii) Grouping sub–graphs into different nodes reduces the interconnections between the nodes of the sub–graphs to a single connection between the new nodes. Figure 1c depicts an example of this case. Proper selection of sub–graphs leads to regular interconnections.

iv) Selecting an equal number of nodes per sub–graph produces new nodes with the same computation time. Nodes with the same computation time maximize the utilization of an array.

Consequently, we choose as criterion for grouping sets of nodes *to collapse sub–paths of identical length into single nodes, selecting them in such a way as to enhance regular interconnections.* Sub–paths of identical length are sub–graphs with nodes at different levels of the graph and with the same number of nodes, meeting the conditions listed above.

The following procedure satisfies the criterion:

- Annotate the nodes with their level in the graph (the level also corresponds to the time when the nodes are executed). This information is obtained by traversing the graph from inputs to outputs.
- Select the length of the sub–paths to collapse according to the desired reduction in the number of nodes in the graph. Such reduction is determined by implementation constraints, namely by the number of nodes possible in the implementation.

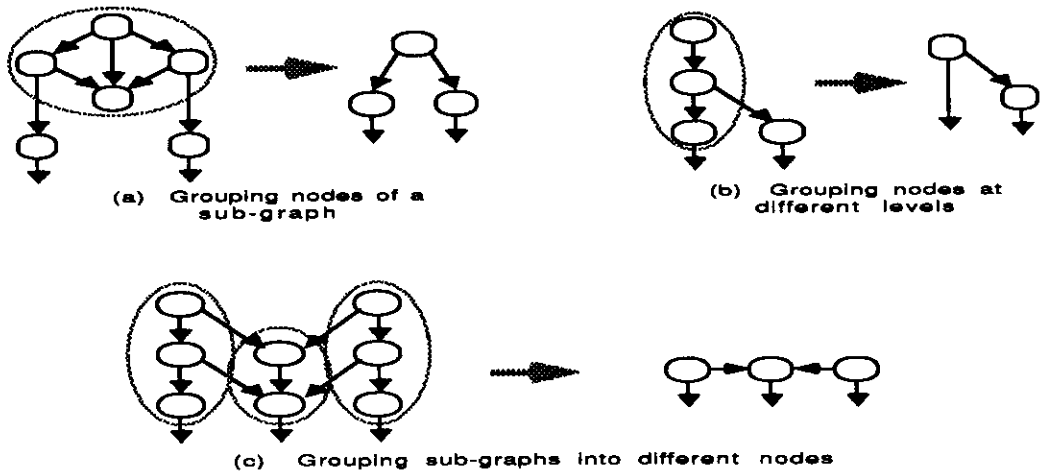
(a) Grouping nodes of a sub-graph
(b) Grouping nodes at different levels
(c) Grouping sub-graphs into different nodes

Figure 1: Options in grouping nodes of the graph

- Select the sub–paths to collapse in such a way as to enhance interconnection regularity.
- Serialize onto the inputs of the new nodes the data used for the different nodes which have been collapsed.

The application of this grouping procedure to some graphs is straightforward. In those cases, it is possible to group sub–paths of identical length without compromising the interconnection structure of the resulting array as shown in Figure 2a. An example of an algorithm with such features is matrix multiplication. However, when sub–paths suitable for collapsing do not have the same length or there are several groupings possible (as is the case with certain algorithms with irregularities), it becomes necessary to evaluate the alternative grouping options. An example of this situation is shown in Figure 2b, where $O(n^2)$ nodes in a graph have to be reduced to $O(n)$. Several groupings are possible. Grouping vertical sub–paths leads to varying computation time per node, while more irregular groupings such as the one shown in the figure achieve the same computation time per node but require a complex interconnection.

In contrast, Figure 2c shows a grouping which does not follow the procedure described above, in particular it doesn't group sub–paths of the graph. This grouping leads to a complex interconnection, higher internal data communications bandwidth, and longer delay (i.e., computation time), undesirable characteristics for an implementation.

Many algorithms have dependence graphs exhibiting varying paths length, such as matrix triangularization, LU–decomposition, inverse of non–singular triangular matrix, QR decomposition, among others. In those cases, the application of the grouping procedure described above will not be succesful, because it is not possible to select groups with the same number of nodes without compromising the interconnection structure of the array. As a consequence, the utilization of the resulting array will be sub–optimal. Utilization degradation is minimized by grouping nodes in such a way that the computation time of each group is similar to the group with the longest computation time. Therefore, we extend the procedure described above to include varying paths length by *performing the selection of sub–paths suitable for collapsing in order of decreasing length.* That is, we select the length of the longest sub–path and perform as many groupings of that length as possible. Then, we perform groupings of length one less than the maximum and continue doing so in decreasing length untill all nodes have been included in some group.

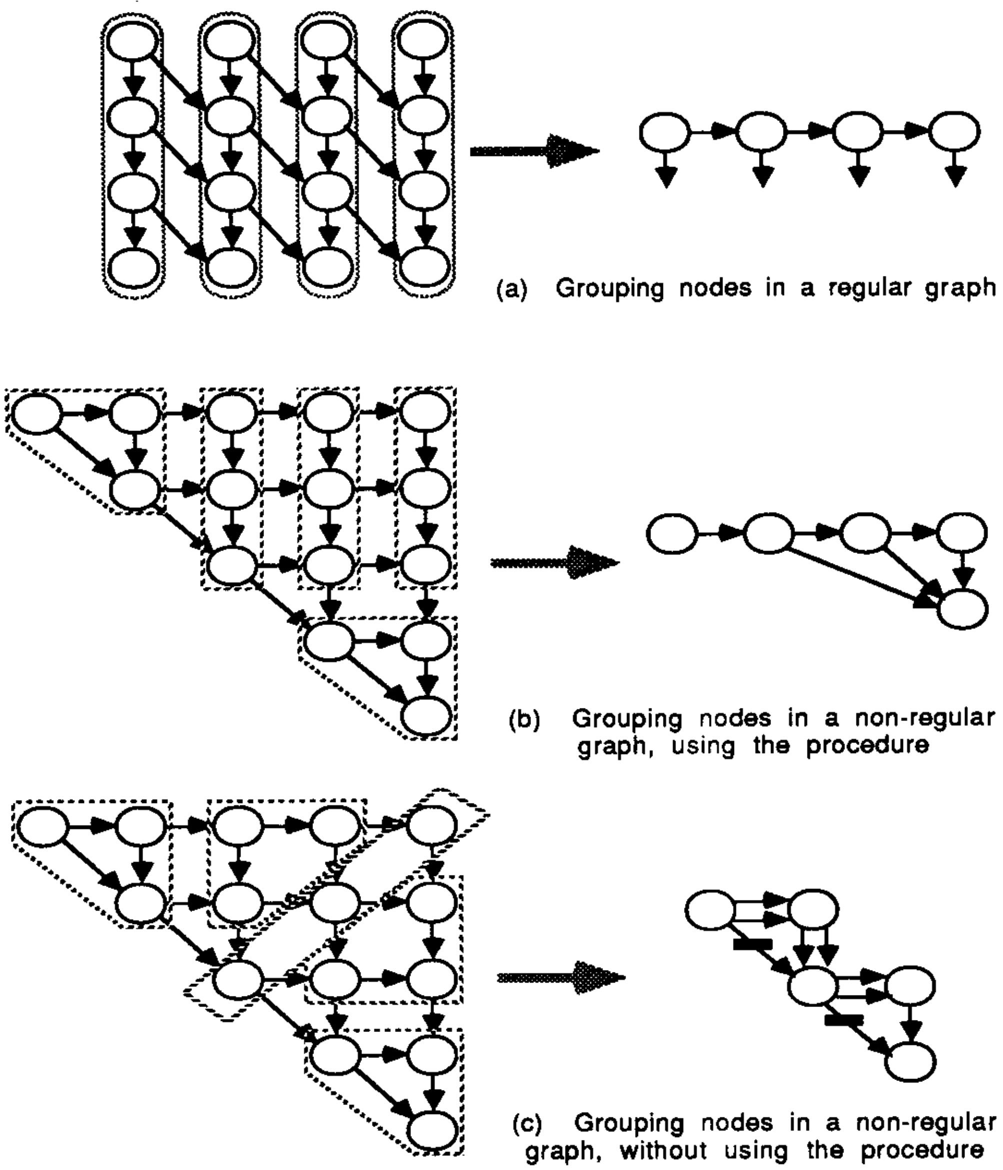
(a) Grouping nodes in a regular graph
(b) Grouping nodes in a non-regular graph, using the procedure
(c) Grouping nodes in a non-regular graph, without using the procedure

Figure 2: Grouping nodes in graphs with constant and varying paths length

We apply now the extended reduction procedure to the LU–decomposition without pivoting.

2.1 Reducing the number of nodes in LU–decomposition without pivoting

Figure 3 shows the fully–parallel dependence graph for the LU–decomposition algorithm without pivoting, for a 6 x 6 matrix, after replacing data broadcasting with data pipelining [13]. In this figure, nodes are annotated with their level in the graph (i.e., the number of nodes traversed from the root of the graph up to and including a node). This graph depicts varying paths length, $O(n^3)$ nodes, and $O(n^2)$ I/O bandwidth. The graph is characterized by sets of sequential operations which are interdependent. Data arrives to the nodes synchronously, without the need to add extra delay nodes. Due to implementation restrictions, this graph is not suitable for implementation: input data elements are needed throughout the graph, implying large input bandwidth, and the graph requires $O(n^3)$ PEs.

We address the requirement of PEs, with the objective of reducing the number of nodes in the graph to $O(n^2)$. For such purpose, we identify sub–paths suitable for collapsing into single nodes according to the criterion stated in Section 2. There are essentially three groupings possible, namely the sub–paths corresponding to rows, columns or diagonals in Figure 3. However, all these sub–paths do not have the same length. Moreover, they exhibit different degree of variation in length. For instance, there is only one diagonal sub–path of length six and eleven diagonal sub–paths of length one, while there are six rows of length six and only one row of length one. Consequently, according to the selection criteria stated in Section 2, we choose to collapse rows (or columns) as shown in Figure 4. A direct implementation of this graph leads to a triangular array.

The alternative of collapsing each diagonal sub–path in Figure 3 into a single node, and mapping the resulting nodes to different cells in a square array, was proposed in [10]. However, the utilization of such array is not adequate because of the variation of the number of operations per cell. For example, the cells in the leftmost column and in the topmost row have only one operation to compute, while the lower rightmost cell has n operations. Consequently, the diagonal sub–paths do not meet the criteria we stated in Section 2, since there is only one sub–path with the longest length. The triangular array derived above has a utilization roughly twice better, for large n, than that of the square array. Therefore, the triangular array performs the computation in the same amount of time with half as many units than the square array.

3 Transformations for Some Algorithm Irregularities

We consider here certain irregular properties which are found in matrix algorithms. These properties complicate the reduction in the number of nodes according to the procedure stated in the previous section. We use the dependence graph of the algorithm, in its fully–parallel or a transformed form, to identify those non–regularities. We propose transformations to the dependence graph to eliminate or reduce the non–regularities, leading to more regular implementations.

We center our attention on the following non–regularities:

a) Broadcasted data flows in more than one direction along some or all of the communication

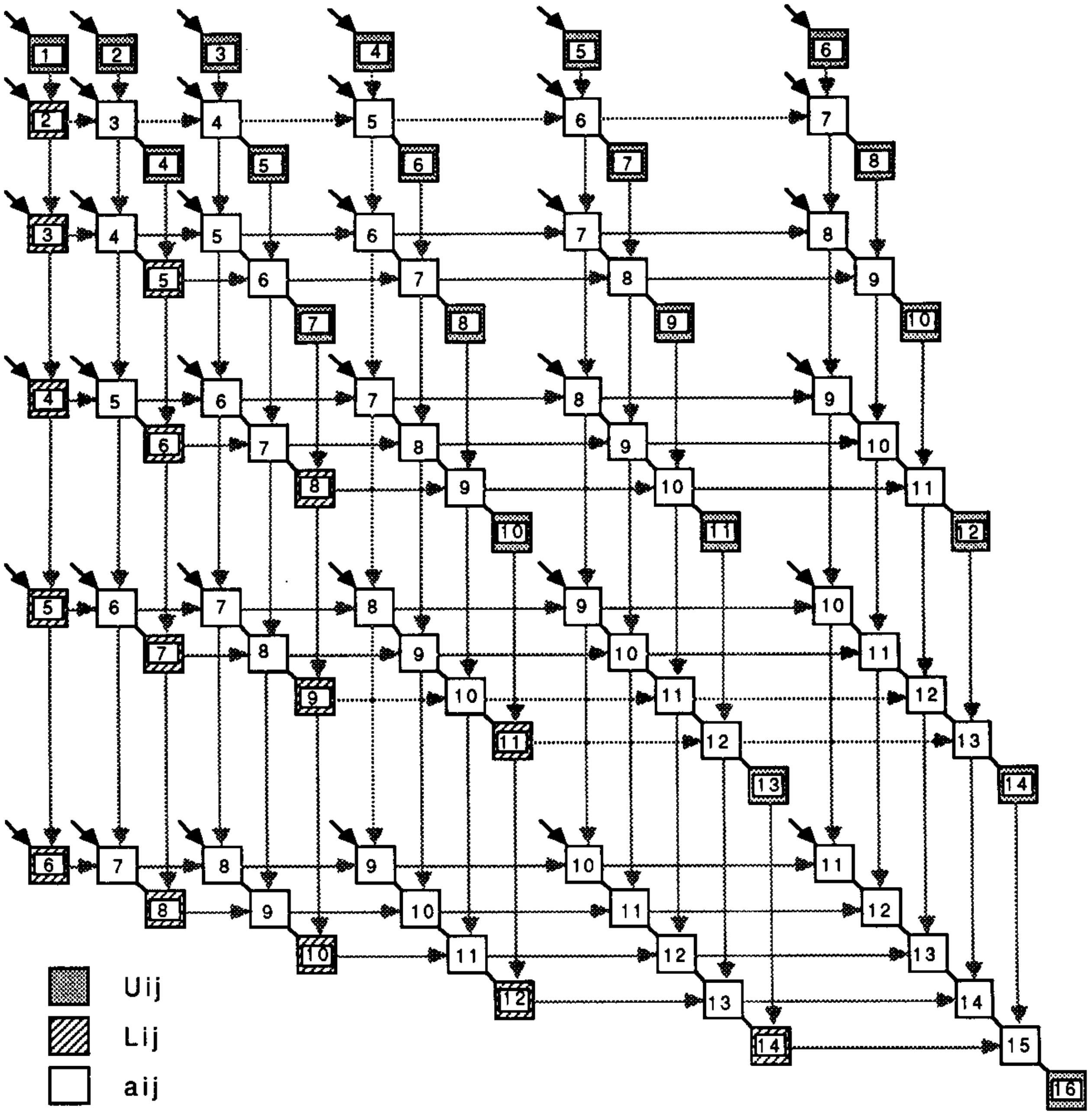

Figure 3: LU–decomposition graph

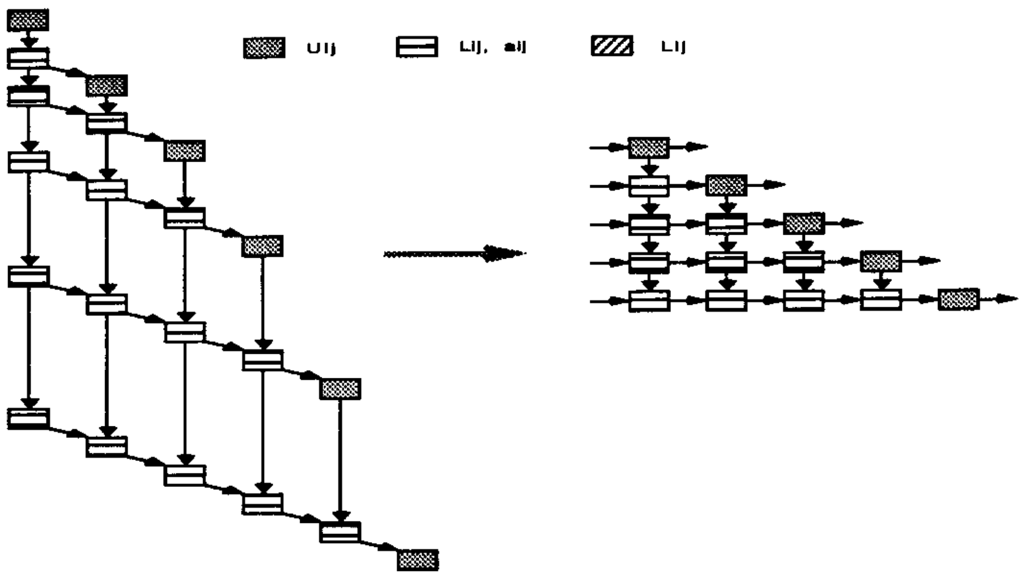

Figure 4: Reducing the number of nodes in LU–decomposition

paths (i.e., horizontal, vertical, diagonal).

b) The interconnection pattern between nodes in the dependence graph is not the same for all nodes.

These non–regularities are described in detail in the following subsections and suitable transformations are proposed for them, so that they can be handled as part of our design methodology. We use LU–decomposition with pivoting to illustrate the existence of the non–regularities and as target for the proposed transformations.

3.1 LU–decomposition algorithm with neighbor pivoting

The LU–decomposition algorithm is an example of Gaussian elimination, which requires division by the diagonal elements of the matrix. Unless the matrix is well conditioned, Gaussian elimination procedures require pivoting for numerical stability. The strategies suggested to cope with this problem are complete or partial pivoting. However, neither of these two schemes is amenable to parallel computation since they require global communications. Gentleman and Kung [5] proposed another scheme, called *neighbor pivoting*, where the pivot is selected as the largest element between two neighbors. They used this approach to devise a systolic array for matrix triangularization. They claim that neighbor pivoting is stable and that numerical experiments have confirmed so. We use this pivoting scheme for LU–decomposition as an example of the type of irregularities that are found in matrix algorithms. We make no specific statements regarding the suitability of this scheme from the numerical point of view.

The LU–decomposition algorithm with neighbor pivoting is described by the dependence graph shown in Figure 5. Broadcasting of intermediate results, originally present in the graph, has been replaced by pipelining according to the methodology described in [13]. In this version of the algorithm, pivots are selected as the largest of a diagonal element and the element in the next row and same column. That is, at iteration k the pivot is chosen as $max(a_{k,k}, a_{k+1,k})$. If the chosen pivot is element $a_{k+1,k}$, rows k and $(k+1)$ must be exchanged.

The graph in Figure 5 exhibits varying paths length, similar to the case without pivoting, but it has the additional irregularity of bi–directional broadcasting. In fact, the selection of the pivot must be broadcasted in both directions in each row: towards the right to compute the remaining elements $u_{i,j}$ $(j > i)$, and towards the left to exchange previously computed elements $l_{i,j}$ $(i < j)$, if necessary. Notice that the arrival of data to the nodes in this graph is not synchronized.

3.2 Transformation of graphs having bi–directional broadcasting with independent paths

We present now a transformation to eliminate bi–directional broadcasting under certain constraints. This type of irregularity is found in matrix algorithms such as square of a matrix and LU–decomposition with neighbor pivoting, among others.

There are two issues of interest in bi–directional broadcasting, namely the synchronization of data and the reduction of nodes in the graph. Both are complicated by bi–directional broadcasting, as we show in Figure 6. In Figure 6a, there is bi–directional broadcasting from the nodes located along the main diagonal of the graph. Nodes to the left of the main diagonal receive data from above before receiving the data broadcasted in a row. In other words, a vertical path reaching a node to the left of the main diagonal in a given row of the graph is shorter than a path going through the diagonal node in the same row. As a consequence, data arrival to the nodes to the left of the main diagonal needs to be synchronized. Such synchronization requires the addition of delay registers along the vertical paths to the left of the source of broadcasting, as shown in Figure 6b. Adding the delays implies inserting new nodes in the graph and the length of the columns is increased by different amounts, so that the graph is not regular anymore (i.e., vertical sub–paths in the graph do not have the same length). As a consequence, the application of the procedure proposed in Section 2 to reduce the number of nodes in the graph leads to low utilization of cells if the grouping is done by columns, or $O(n)$ storage requirements in the cells if rows (including the delays) are collapsed into single nodes.

The problems outlined above can be eliminated for the particular case in which all computation paths to the left (or right) of the source of broadcasting are independent (i.e., they are disjoint), as it is the case in Figure 6a. In such a case, it is possible to move the independent computation paths to the other side of the node source of broadcasting, so that the bi–directional data flow is eliminated. The result of this type of transformation is shown in Figure 6c.

We apply now the transformation outlined above to the algorithm for LU–decomposition with pivoting, which exhibits this type of bi–directional broadcasting. In Figure 5, the sources of bi–directional broadcasting are the nodes used to compute the elements $u_{i,i}$. The broadcasted data is the signal indicating whether the corresponding rows have to be exchanged or not, depending on the pivot selected. The computation paths which perform the exchange of elements $l_{i,j}$ (i.e., the vertical computation paths to the left of nodes computing $u_{i,i}$) are independent. Therefore, as indicated above, it is possible to move these independent paths to the right of the source of broadcasting in such a way that the bi–directional data flow is transformed into uni–directional. This transformation is shown in Figure 7. The resulting graph has broadcasting in only one

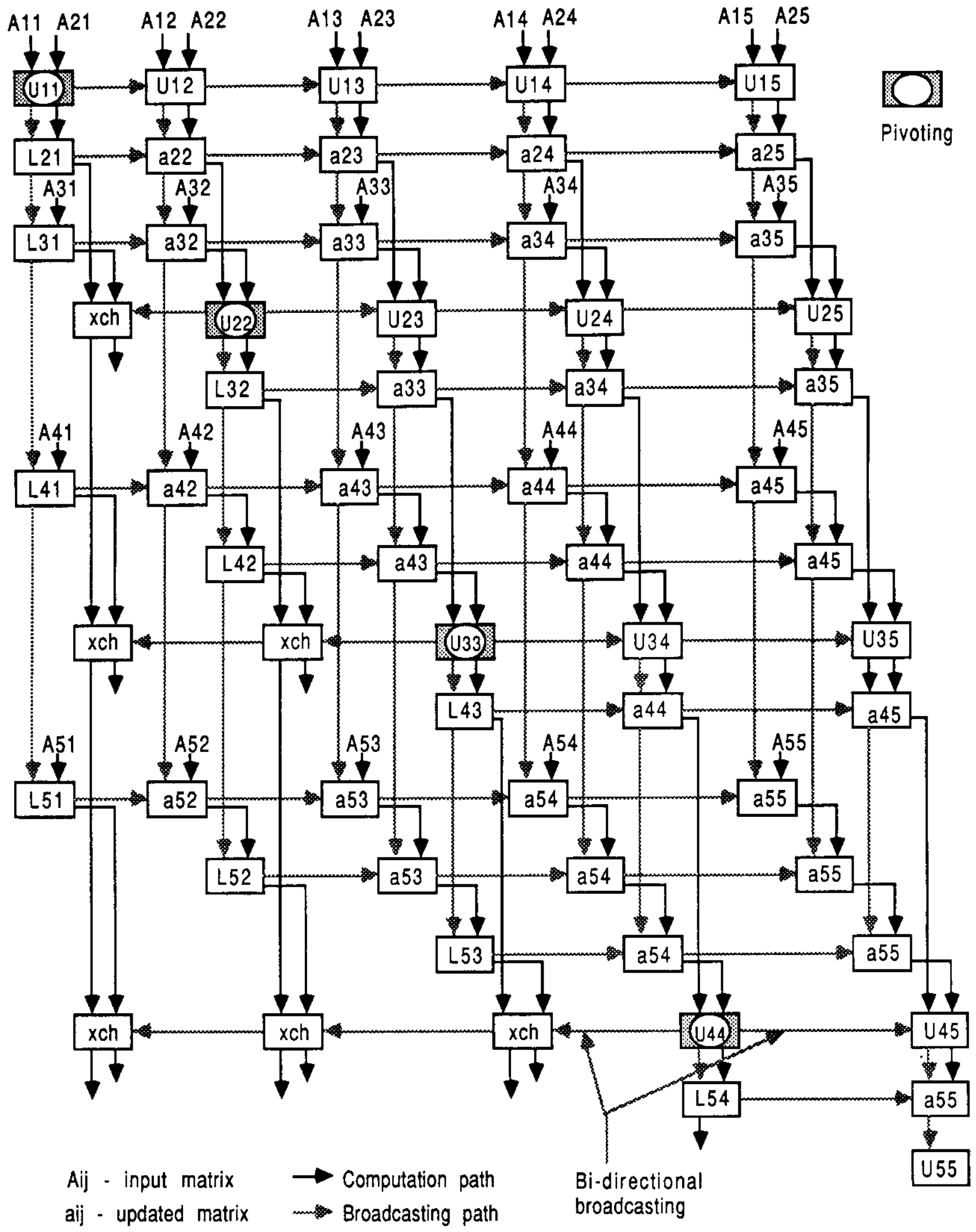

Figure 5: LU–decomposition with neighbor pivoting

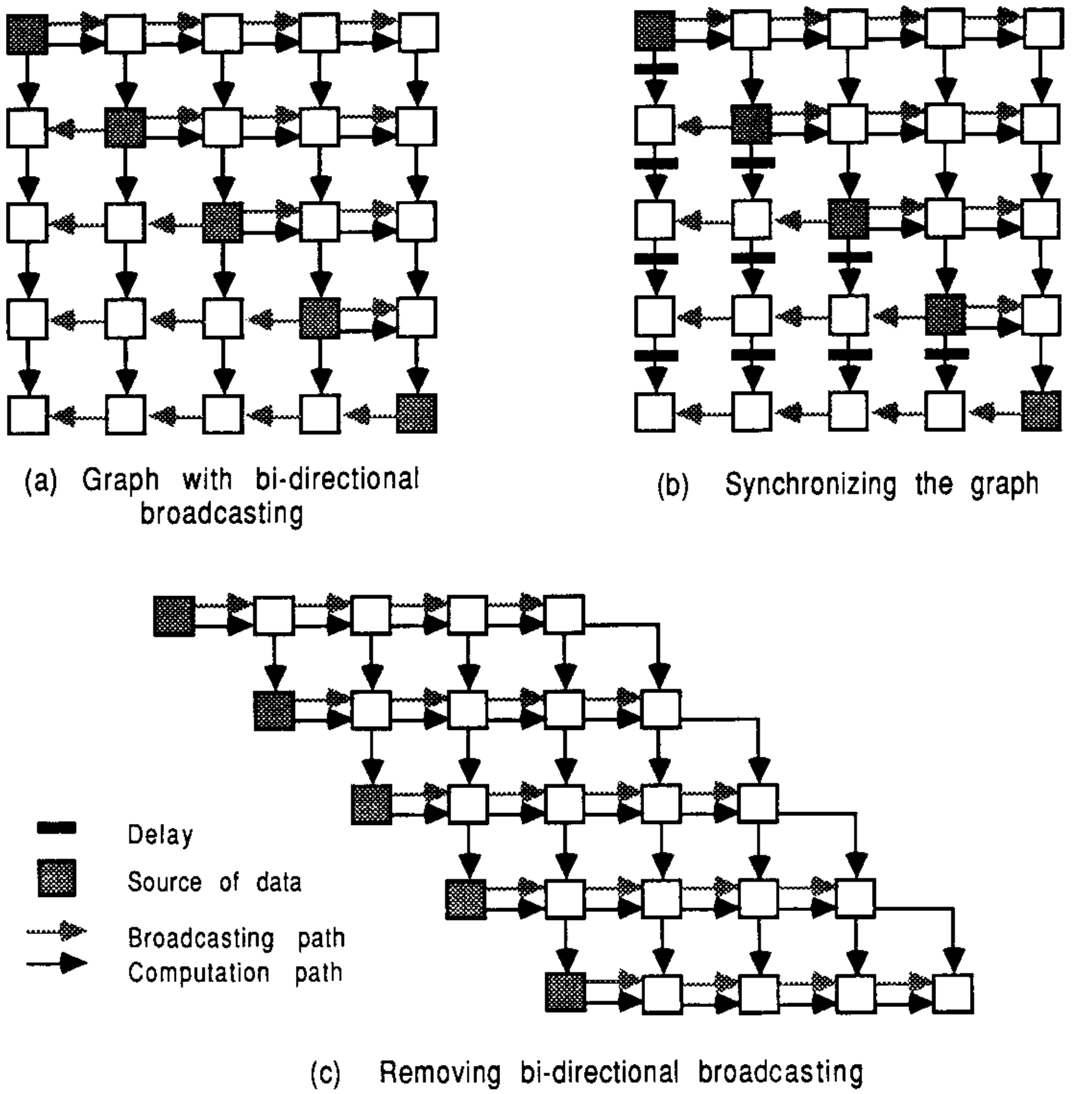
(a) Graph with bi-directional broadcasting
(b) Synchronizing the graph
Delay
Source of data
Broadcasting path
Computation path
(c) Removing bi-directional broadcasting

Figure 6: Independent computation paths with bi–directional broadcasting

direction.

However, the interconnection pattern between nodes of the graph in Figure 7 is non–uniform, as shown by the shaded region. The interconnection of nodes computing updated values of the matrix is different than the interconnection of nodes to exchange the elements $l_{i,j}$. This is not surprising, since they are entirely different operations. In addition, the number of nodes in the graph is $O(n^3)$. The transformation described in Section 2 to reduce the number of nodes cannot be easily applied here, because of the non–uniform interconnection pattern present at the boundary of the graph. Such transformation is based upon selecting sub–paths of the graph and collapsing them into single nodes. The irregular interconnection complicates such selection, or originates complex interconnections.

For instance, if we want to collapse rows of the graph as done before, we find that the parts of the rows exchanging the elements $l_{i,j}$ are connected to many other rows (i.e., nodes to exchange elements $l_{i,j}$ in one row are connected to $O(n)$ rows of the graph). This implies a large fan–in for the collapsed nodes exchanging the elements $l_{i,j}$. Moreover, pipelined flow of elements $l_{i,j}$ and $u_{i,j}$ towards the exchange nodes requires the synchronization of data arrival to those nodes, task which is not straightforward with the graph as shown. Consequently, given the irregular interconnection pattern, it is not possible to collapse rows of the graph as done previously for the case without pivoting. In addition, attempting to partition the graph into a regular and a non–regular portions, grouping the nodes in them separately and combining the resulting parts, faces the same problems just mentioned. Further transformations to solve the non–regular interconnection pattern are discussed next.

3.3 Transforming non–uniform interconnection pattern

As illustrated by the previous example, another type of irregularity in matrix algorithms occurs when nodes in the dependence graph have non–uniform interconnection pattern. That is, nodes are interconnected with a regular pattern excepting some interconnections at the boundaries of the graph which are not the same. An example of this kind of irregularity is shown in Figure 8a. This non–uniform interconnection pattern complicates the collapsing of sets of nodes into single nodes, because sub–paths in the graph do not have the same structure.

Our approach to solve the non–uniform interconnection pattern problem consists of replacing the irregularity with a regular interconnection. To achieve this, we add delay registers to the irregular boundaries in such a way that their interconnection corresponds to the same regular pattern as the one existing for the rest of the nodes. The way such delay registers are added depends on the interconnection structure of the nodes whose non–uniformity is being solved. An example of transforming non–uniform interconnections is depicted in Figure 8b. As shown in the figure, this transformation might increase the total computation time of the algorithm (i.e., delay), but the throughput is not affected. For algorithms which are computed for many instances, this increase in delay is not significant.

We have found this approach to be suitable for some algorithms, though we can't claim that it is always possible to do so. In other words, this is an example of a transformation adequate for certain algorithms, but not necessarily all cases.

We apply now the transformation proposed above to the non–regular interconnection pattern

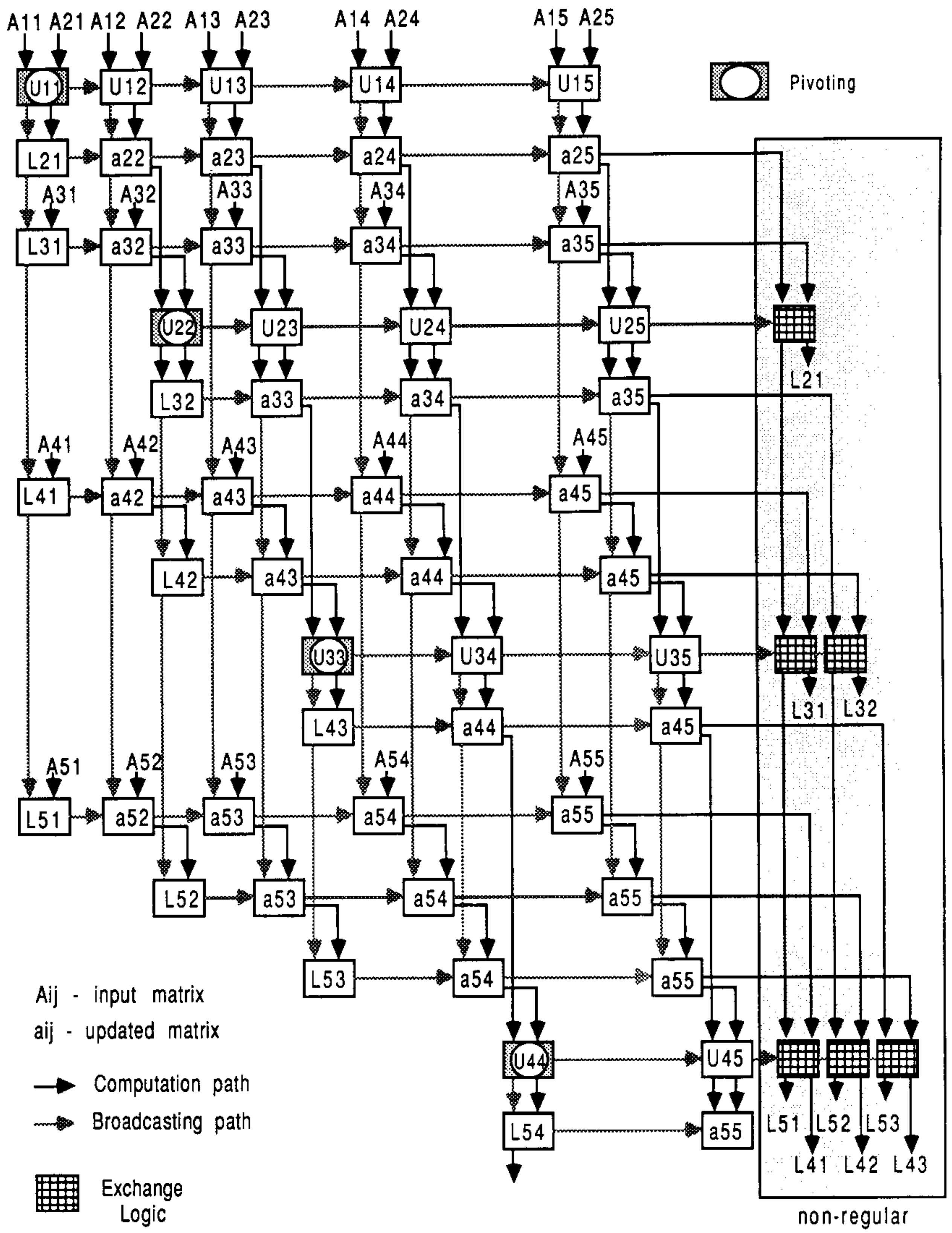

Figure 7: Transforming bi-directional broadcasting in LU-decomposition with neighbor pivoting

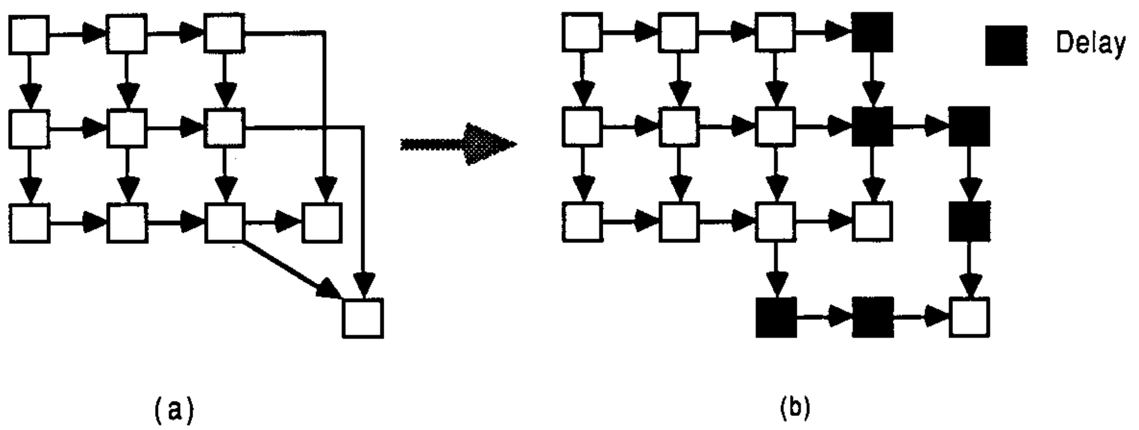

Figure 8: Non–uniform interconnection pattern

existing in the transformed graph for LU–decomposition with pivoting shown in Figure 7. We transform the irregular pattern into a regular one by adding delay registers to the boundary paths, namely the ones used to exchange the elements $l_{i,j}$. These registers are connected with the same structure as the other nodes, as shown in Figure 9. The arrival of data to the nodes of this graph has also been synchronized. It turns out that the task is accomplished without difficulty, resulting in a graph with a regular interconnection pattern.

The new graph exhibits varying paths length but it is suitable for grouping nodes. We apply now the procedure proposed for such task in Section 2 and group rows (or columns) of the graph into single nodes. The graph resulting from such grouping, shown in Figure 10, is regular and can be mapped directly into a triangular array.

4 Conclusions

We have presented a procedure to reduce the number of nodes in the fully–parallel dependence graph of matrix algorithms, with the purpose of reducing the number of cells needed in an implementation. Such procedure is part of a design methodology for arrays of PEs that we are researching. This methodology consists of the application of transformations on a fully–parallel graph describing the algorithm, to fulfill implementation restrictions. We have also identified some irregularities which complicate the proposed procedure. These irregularities are varying paths length, bi–directional broadcasting and non–regular interconnection pattern between nodes of the dependence graph. We have proposed an extension to the procedure for reducing the number of nodes, to take into account the varying path length. Moreover, we have proposed transformations to the dependence graph to remove bi–directional broadcasting and non–regular interconnection pattern under certain constraints. The graphs obtained as a result of the proposed mechanisms are suitable for further transformations aimed towards regular algorithms, or for direct implementations as arrays of PEs.

We have used the LU–decomposition, without and with pivoting, as examples of application of our methodology to algorithms exhibiting the non–regularities considered. Such application has resulted in triangular arrays for these algorithms, with better utilization than the square arrays formerly proposed for it.

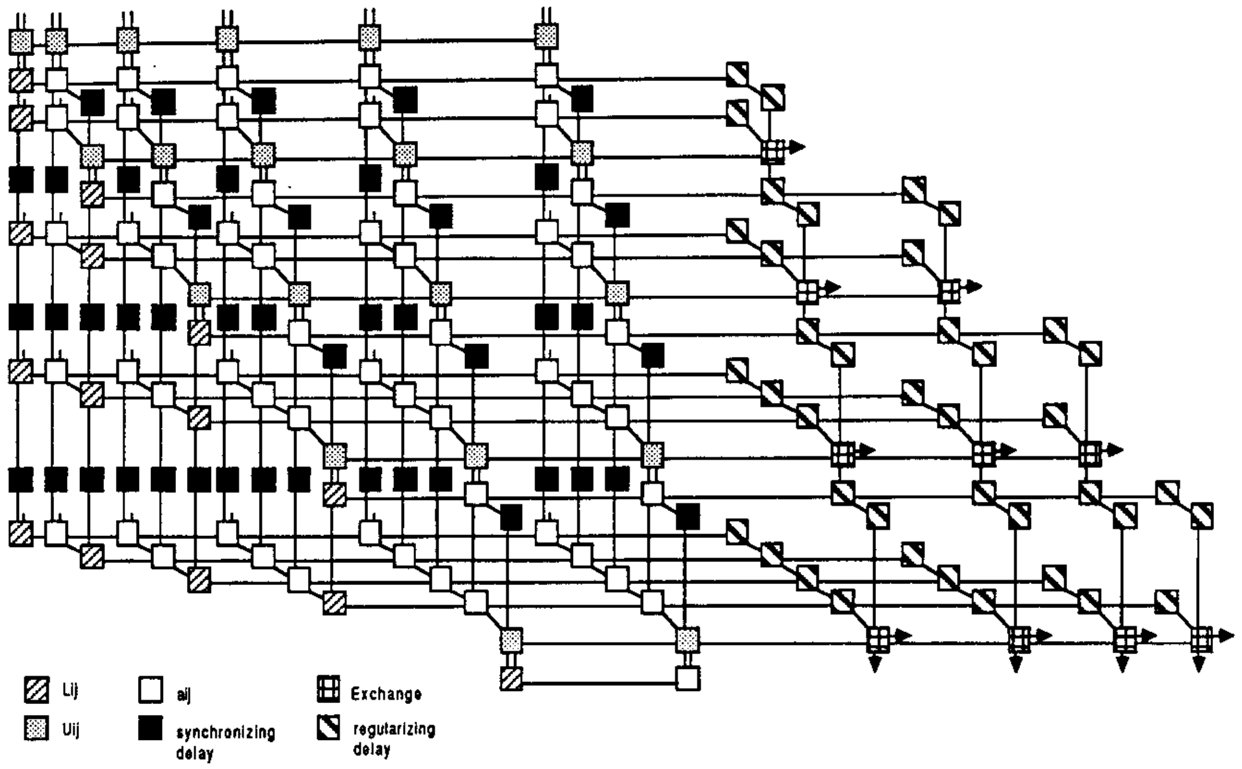

Figure 9: Transforming non–regular interconnections in LU–decomposition with neighbor pivoting

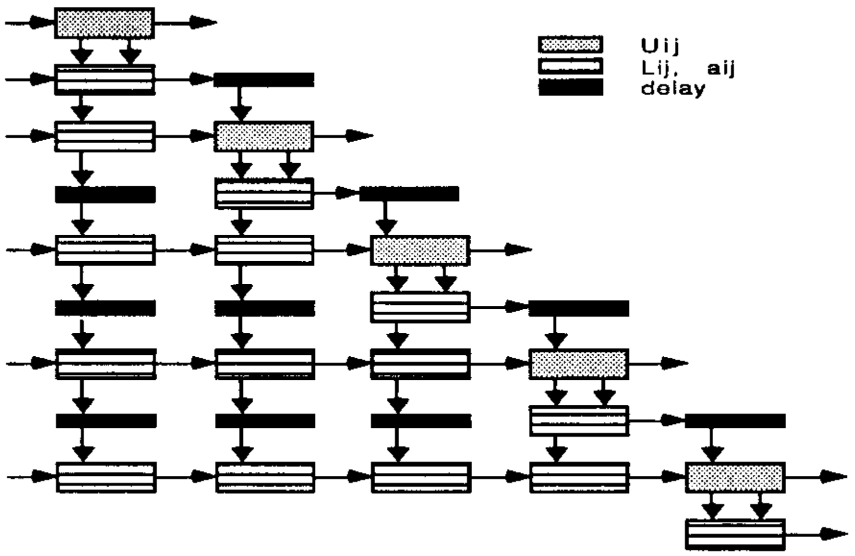

Figure 10: Regular (triangular) array for LU–decomposition with neighbor pivoting

Our research has permitted us to describe the features of a few basic transformations [12], [13]. Their application has allowed us to show that it is possible to incorporate implementation restrictions, and now specific irregularities, as part of a design methodology. However, these transformations are not an exhaustive collection for all possible irregularities and matrix computations. In addition, as shown here, there are cases where the details of a certain transformation are specific to a given algorithm. The objectives of our current research include the identification and formal definition of a larger set of transformations, for a more varied class of matrix algorithms. The ultimate goal of the proposed research is to provide the designer with a collection of transformations which are applied to a target algorithm. In this way, the design process becomes a search, in the space of solutions available through the transformations, for the alternative which offers the best cost–performance trade–offs.

References

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

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

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

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

[6] J. Speiser and H. Whitehouse, "Parallel processing algorithms and architectures for real–time signal processing," in *SPIE Real–Time Signal Processing IV*, pp. 2–9, 1981.

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

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

[9] S. Kung, S. Lo, and P. Lewis, "Optimal systolic design for the transitive closure and the shortest path problems," *IEEE Trans. Computers*, vol. C-36, pp. 603–614, May 1987.

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

[11] 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.

[12] 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.