# REMOVING ALGORITHM IRREGULARITIES IN THE DESIGN OF ARRAYS FOR MATRIX COMPUTATIONS

Jaime H. Moreno
Tomas Lang

August 1987
CSD-870040

# Removing Algorithm Irregularities in the Design of Arrays for Matrix Computations

Jaime H. Moreno, Tomás Lang
Computer Science Department
University of California, Los Angeles

Report No. CSD–870040*
August 1987

*This research has 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 address some irregularities in matrix algorithms, as part of a systematic design methodology for arrays of processing elements (PEs) that we are researching. We propose a procedure to reduce the number of nodes in the fully–parallel dependence graph of matrix algorithms. We identify some irregularities which complicate such reduction of nodes and propose transformations to the dependence graph to remove them. The graphs obtained from such transformations are suitable for further transformations aimed towards regular algorithms or for direct implementation as arrays of PEs. The irregularities considered are bi–directional broadcasting and non–regular interconnection pattern between levels of the dependence graph. We use LU–decomposition, without and with pivoting, as examples of application of our transformations. The methodology results in triangular arrays for this computation, with better utilization than 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, but they might be non–optimal for a particular algorithm. Some transformational methodologies have been proposed [2], which either restrict the form of the algorithm (i.e., a recurrence equation) or are unable to incorporate certain implementation restrictions, such as number of I/O pads, limited data broadcasting, or lower bound on efficiency.

We are researching a general and systematic design methodology for matrix algorithms, with the capability to handle and relate features of the algorithm and the implementation in a unified manner [3]. We have applied a preliminary version of this methodology to the algorithms for matrix multiplication and LU–decomposition [4]. This methodology is based on the dependence graph of the algorithms and provides mechanisms to deal with issues such as data broadcasting, data synchronization, interconnection structure, I/O bandwidth, number of PEs, throughput, delay, and utilization of PEs. Starting from a fully–parallel graph, in which nodes represent the operations and edges correspond to data communications, we apply transformations to the graph to incorporate the issues indicated above. Since the proposed methodology addresses multi–instance and single–instance computations, some transformations exploit pipelining of data to enhance concurrency and reduce communication requirements, while other transformations are oriented to reduce the computation time.

Our previous work has considered algorithms which, in general, exhibit regular properties. In this paper, we address algorithms which exhibit irregularities, such as square of a matrix, LU–decomposition without and with pivoting, matrix triangularization, inverse of non–singular triangular matrix, among others. In Section 2, we briefly review our design methodology. In Section 3, we discuss the issues involved in reducing the number of nodes in a graph, when such number is large, and propose a procedure to perform such reduction. In Section 4, we apply the proposed procedure to the algorithm for LU–decomposition without pivoting. In Section 5, we identify some algorithm irregularities that complicate the design of arrays for matrix computations, in particular irregularities that complicate the reduction of the number of nodes in the dependence graph. We describe some transformations to the graph to remove such irregularities and illustrate the use of these transformations by applying them to the algorithm for LU–decomposition with neighbor pivoting.

# 2 A Graph-Oriented Design Methodology for Arrays of PEs

The development of systematic methodologies for the design of arrays has been actively pursued recently. In [2], Fortes et al. review seventeen different methods for the design of algorithmically specified systolic arrays. New methods have been proposed after that review. Fortes et al. conclude

that the most common characteristic of the proposed methods is the use of a *transformational approach*, 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 have limitations. The methods are, in general, unable to incorporate implementation restrictions, such as number of I/O pads, suitable utilization of processing elements, or limited data broadcasting. Furthermore, the algorithms considered are restricted to belong to limited classes, such as those representable as uniform recurrence equations, program with loops, or restricted types of dependence graphs.

We have proposed a graph–oriented systematic design methodology for arrays of PEs [3]. 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. These graphs are characterized by having all inputs and outputs available in parallel and no loops (i.e., loops are unfolded). The graph doesn't include synchronization of the arrival of data to the nodes, since such synchronization is accomplished as a result of the methodology. Such dependence 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 optimal throughput (for multi–instance computations), but may require complex interconnection structure, high I/O bandwidth, and large number of units. We deal with these problems, while still attempting to preserve the delay and throughput inherent in the dependence graph.

Our methodology consists of applying an ordered sequence of transformations to the graph, to incorporate implementation restrictions. The specific transformations depend on the particular parameters of interest. At each step in this sequence, we obtain a new graph (i.e., a new form of the algorithm), which includes some implementation restriction. The final step in the sequence consists of a direct mapping of the transformed graph onto an array of PEs.

Applications of a preliminary version of our methodology have used transformations which incorporate a subset of the issues arising in a design. Although some of those transformations are of general application, others are appropriate only for specific cases. We have also shown that the application of different transformations to the dependence graph of an algorithm may lead to entirely different architectures. Our current research is oriented towards identifying a general set of transformations and providing a formal definition for them.

We summarize now our methodology using matrix multiplication as an example. The details of applying the methodology to this algorithm are found in [3], [4]. The dependence graph for multiplication of square matrices is shown in Figure 1. It basically consists of a collection of $n^2$ inner–product trees. From the graph, we can infer the following properties of the algorithm:

- Each input data element is used in $n$ inner–product trees. This indicates that broadcasting of input data is required.
- Inner–product trees are independent among themselves; they depend only on the input data.
- A pipelined implementation of the graph has delay $O(logn)$ and throughput $T = 1[eval/cycle]$. Such implementation requires $O(n^2)$ I/O bandwidth, broadcasting of the input data, and $O(n^3)$ PEs.

Figure 2 shows some alternative paths available in the design of arrays for matrix multiplica-

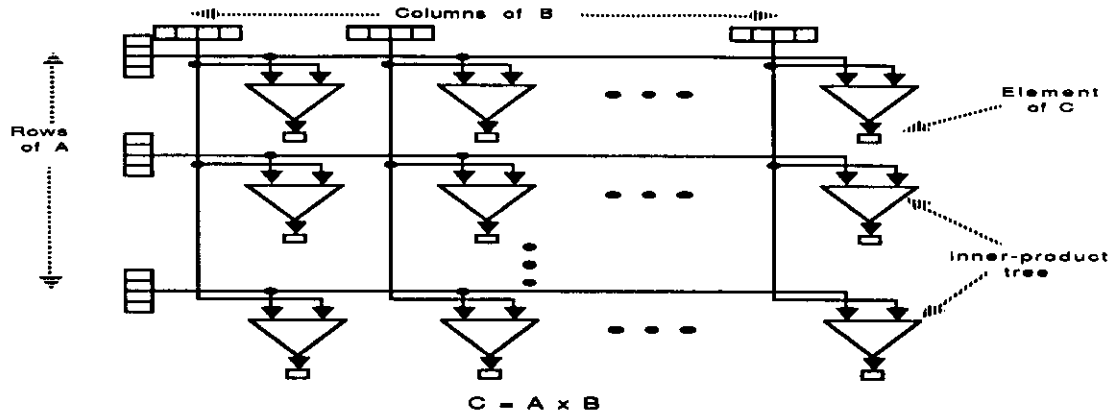



Figure 1: Matrix multiplication dependence graph

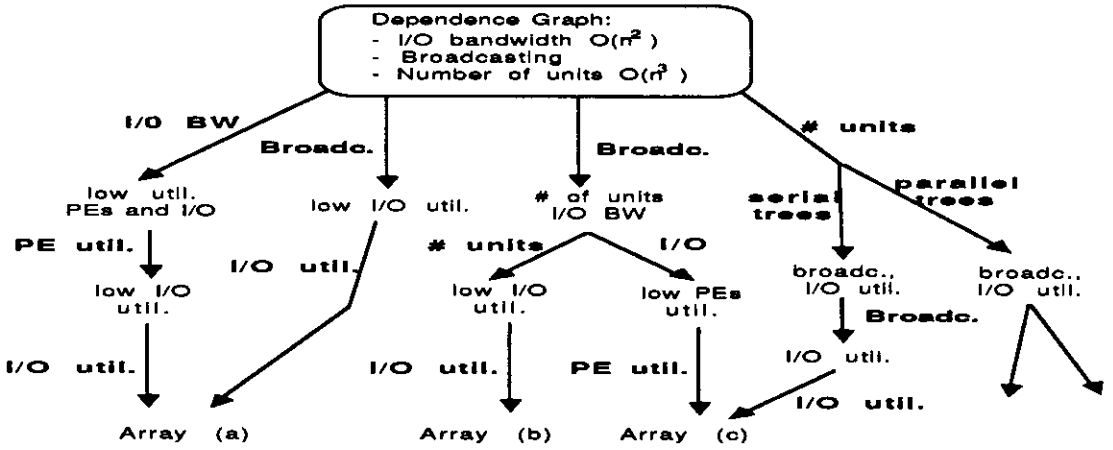


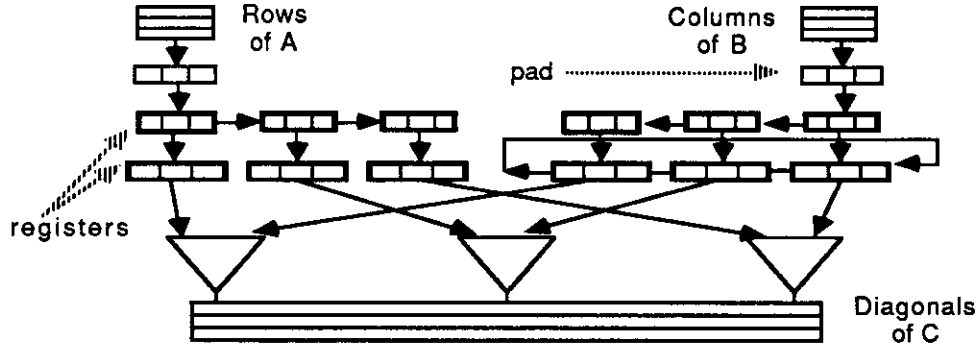

Figure 2: Alternatives in the design of arrays for matrix multiplication

tion, depending on which implementation restrictions, and in which order, are considered. This is reflected on the sequence of transformations applied to the dependence graph. Each label in this figure indicates the implementation constraint addressed by a transformation. Different arrays are obtained with different sequences of transformations, although some paths lead to identical arrays. The arrays obtained from the transformations outlined in Figure 2 are shown in Figure 3.

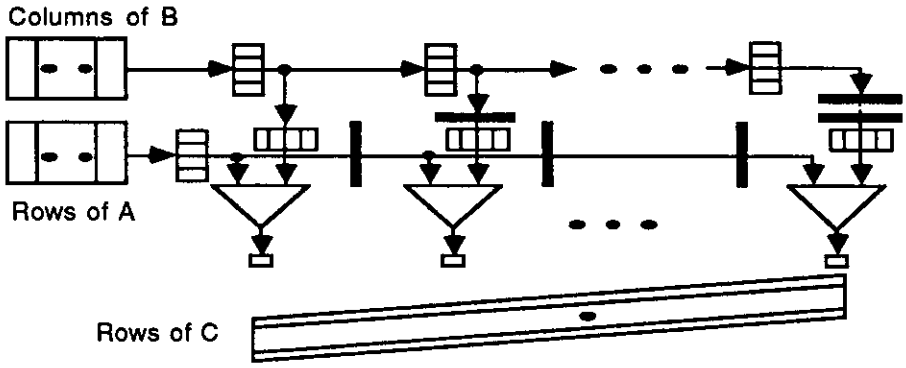
Considering inner–product as an elementary operation, matrix multiplication exhibits the following characteristics:

- The same sub–algorithm (i.e., inner–product) is applied to different sets of data.
- All paths in the fully–parallel graph have the same length.
- The interconnection pattern is the same for all instances of the sub–algorithm.

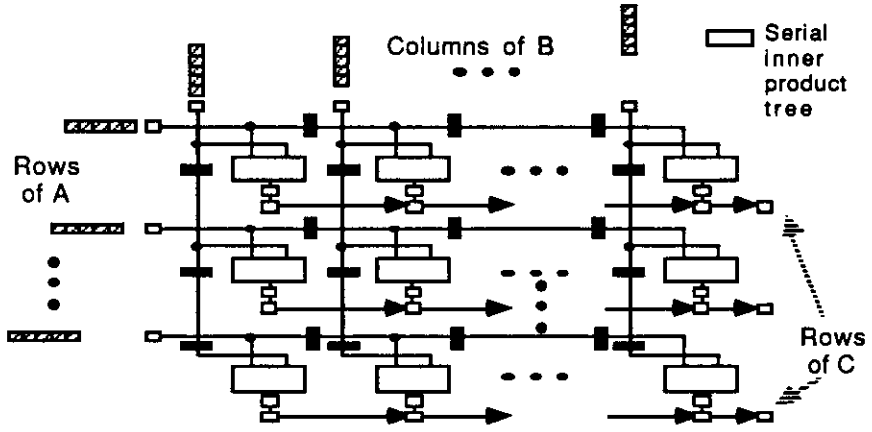
The regular characteristics listed above make it possible to apply transformations to the graph in a straightforward manner. However, regularity is not present in all algorithms of interest. There are cases in which certain deviations exist that render the design of arrays more complex. In particular, reducing the number of nodes in a graph is complicated by algorithm irregularities. In the next section we discuss a procedure to perform the reduction of the number of nodes in




(a) Multiple tree architecture




(b)Linear array of Trees




(c) Square array

Figure 3: Arrays for matrix multiplication

a graph. Later, we identify some algorithm irregularities that complicate the application of the procedure and suggest transformations to the dependence graph that remove the irregularities. The resulting graphs are suitable for other transformations, aimed towards regular graphs, or for direct implementation as arrays of PEs.

# 3 Reducing the Number of Nodes in the Dependence Graphs

The fully–parallel graph of a matrix algorithm 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 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.

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

- Obtain new nodes with similar computation time, so that the utilization of an array is maximized.
- Obtain a regular interconnection pattern among the new nodes.

To fulfill these constraints we have to consider the following issues:

i) Grouping nodes of a sub–graph reduces the interconnection requirements, since the dependences between the nodes in a group are transformed into dependences within the new node. That is, no communication path is required for those nodes. An example of this situation is shown in Figure 4a.

ii) Grouping nodes at different levels in the graph (i.e., nodes which are computed at different time steps) preserves the delay of the computation. This is in contrast to grouping nodes at the same level, because in such a case the path length is increased, eventually increasing the length of the longest path in the graph. Grouping nodes at different levels is shown in Figure 4b.

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 4c 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.

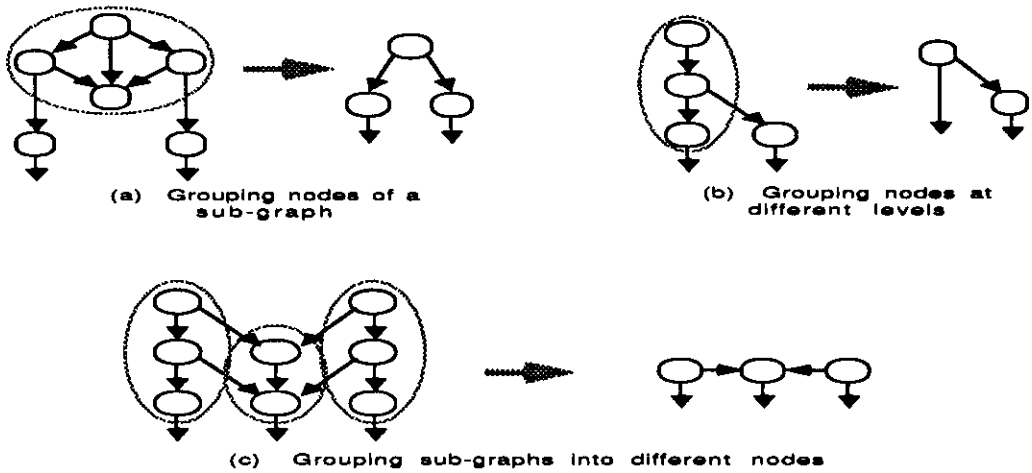



Figure 4: Options in grouping nodes of the graph

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 used). This information is obtained by traversing the graph from inputs to outputs. We assume the same computation time for all nodes.
- Select the length of the longest sub–path according to the desired reduction in the number of nodes in the graph.
- Select sub–paths for collapsing to enhance interconnection regularity and maximize utilization. Since high utilization is obtained when the computation time of nodes is close to the longest node, sub–paths are selected in decreasing length.
- Serialize onto the inputs of the new nodes the data used for the different nodes which have been collapsed.

The application of this grouping criterion to regular algorithms such as matrix multiplication is straightforward. In such a case, it is possible to group sub–paths of identical length without compromising the interconnection structure of the resulting array. An example is shown in Figure 5a. However, when sub–paths suitable for collapsing do not have the same length or there are several groupings possible (as it 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 5b, where $O(n^2)$ nodes in a graph are 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 they require a complex interconnection. The selection of any of those alternatives is based on which performance measure is most relevant, such as utilization or interconnection of the cells in an array.

Figure 5c shows a grouping which does not follow the procedure described above. This grouping leads to a complex interconnection, higher bandwidth, and longer delay (i.e., computation time), undesirable characteristics for an implementation.

In the next Section, we apply the reduction procedure to the LU–decomposition without pivoting, an example of an irregular algorithm.

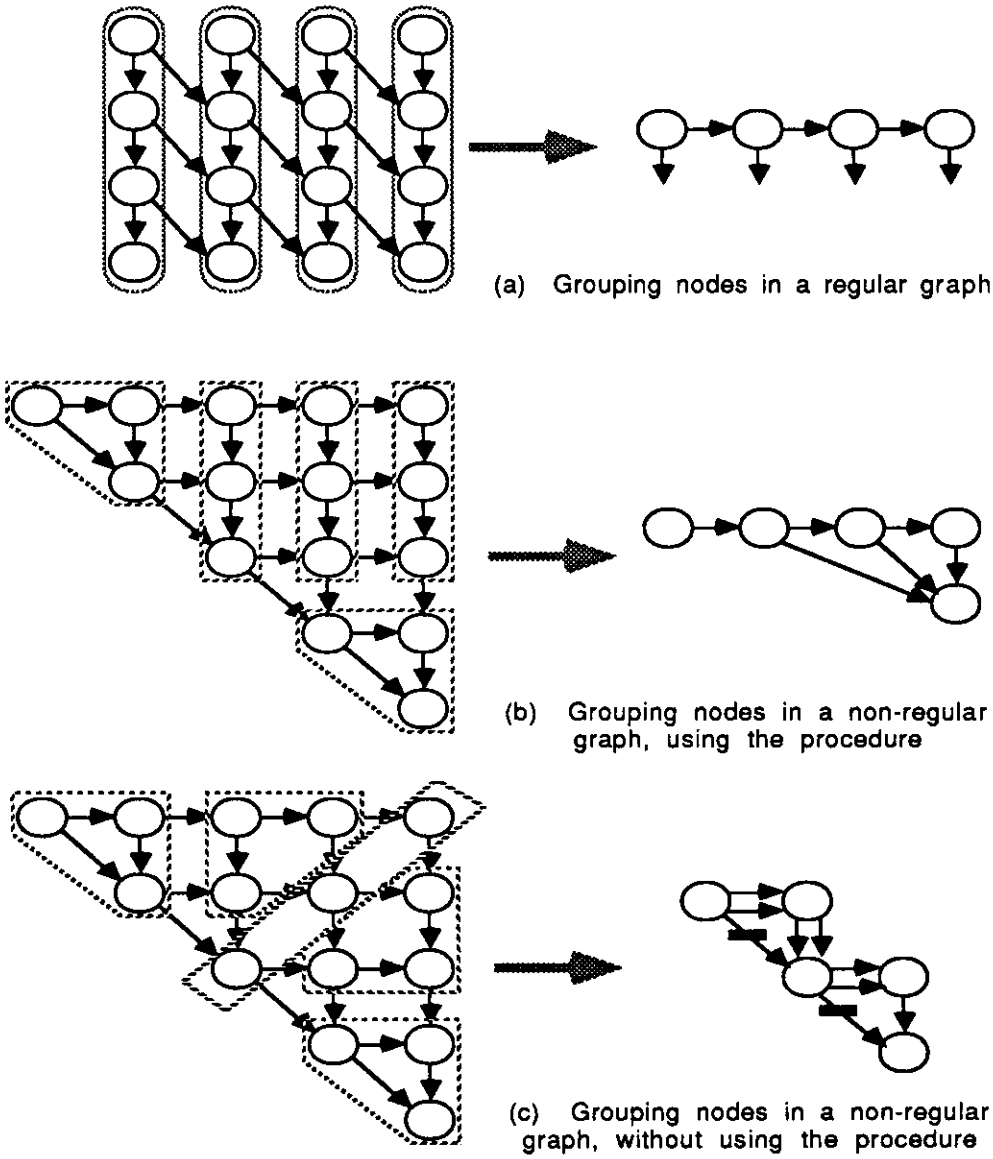



Figure 5: Grouping nodes in graphs with constant and varying parallelism

# 4 LU–decomposition algorithm without pivoting

The LU–decomposition without pivoting for a 6 x 6 matrix is described by the dependence graph shown in Figure 6. In this graph, there is broadcasting of intermediate results. We solve the broadcasting problem by replacing it with data pipelining, using the methodology described in [3], [4]. The graph resulting from the application of such transformation is shown in Figure 7, which also indicates the level of the nodes in the graph. The graph has $O(n^3)$ nodes and is characterized by sets of sequential computations which are interdependent. An attractive result of the transformation applied is that data arrives to the nodes synchronously, without the need to add extra delay nodes. Next, we reduce the number of nodes in this graph by applying the procedure described in the previous section.

## 4.1 Reducing the number of nodes in LU–decomposition without pivoting

Due to implementation restrictions, the transformed graph for the LU-decomposition without pivoting shown in Figure 7 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 first, with the objective to reduce 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 3. There are essentially three groupings possible, namely the sub-paths corresponding to rows, columns or diagonals in Figure 7. 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 3, we choose to collapse rows (or columns) as shown in Figure 8. A direct implementation of this graph leads to a triangular array.

The alternative of collapsing each diagonal sub–path in Figure 7 into a single node, and mapping the resulting nodes to different cells in a square array, was proposed in [6]. However, the utilization of such array is not adequate because of the variation of the number of operations per cell. For example, the top leftmost cell has 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 3, 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.

# 5 Transformations for Algorithm Irregularities

We consider here certain irregular properties which are found in matrix algorithms. These properties essentially complicate the reduction in the number of nodes and synchronization of nodes in a graph. We use the dependence graph of the algorithm, in its fully–parallel or a transformed form, to identify those non–regularities. Since we advocate a graph–oriented design methodology for arrays of PEs, we propose transformations to the dependence graph to include implementation restrictions and to eliminate or reduce the non–regularities, leading to more regular implementations.

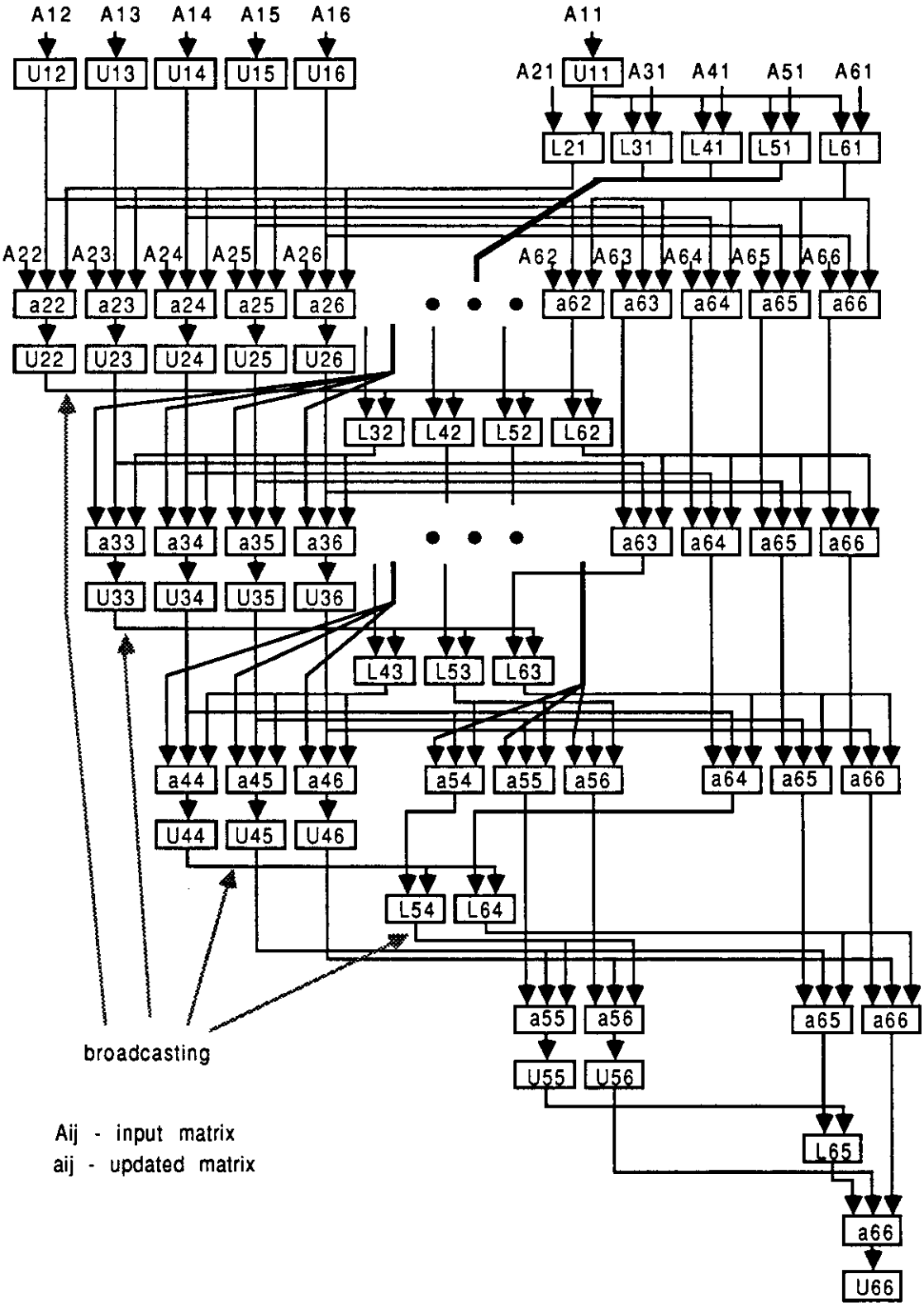



Figure 6: LU–decomposition graph

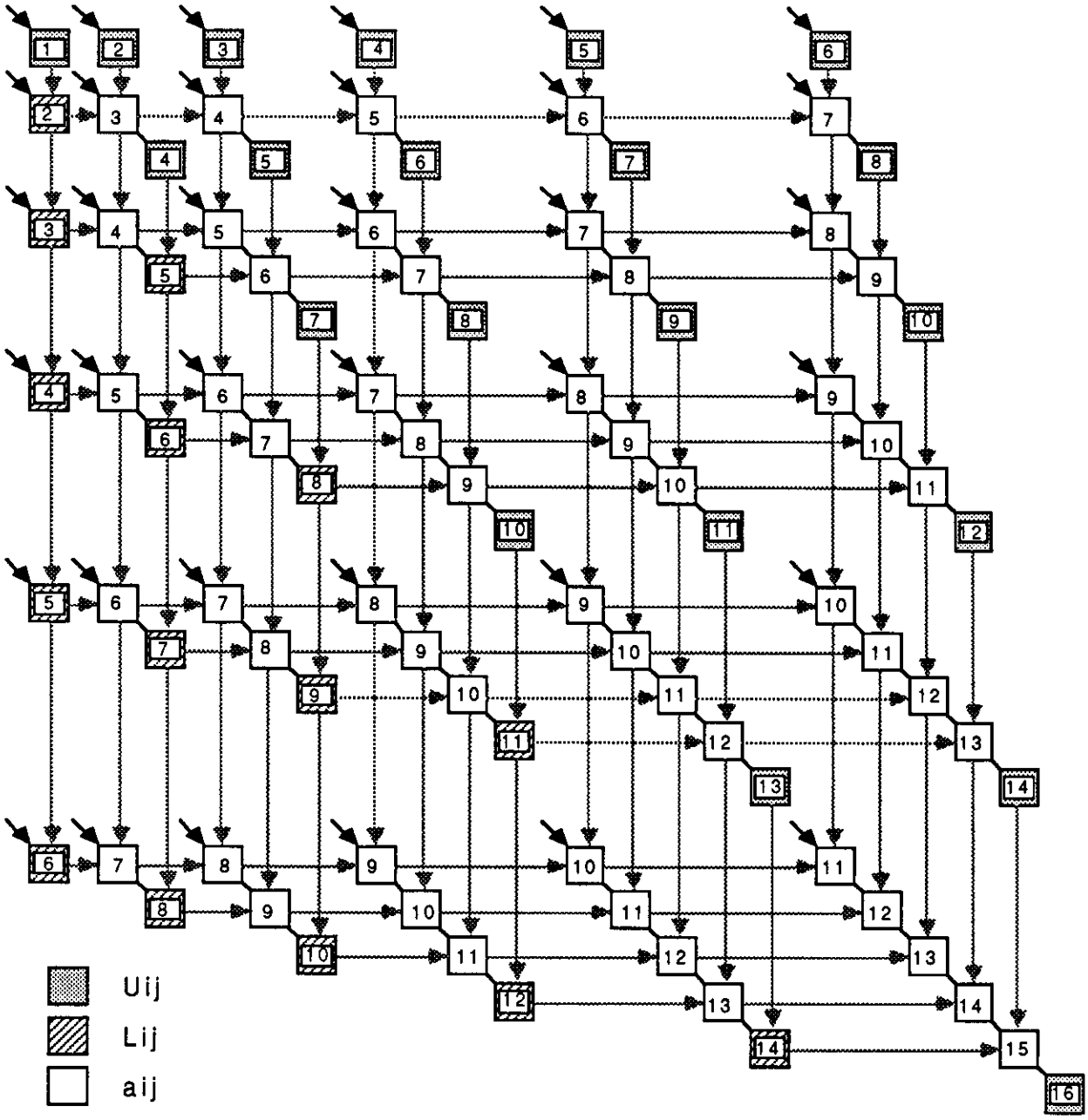



Figure 7: LU–decomposition graph without data broadcasting

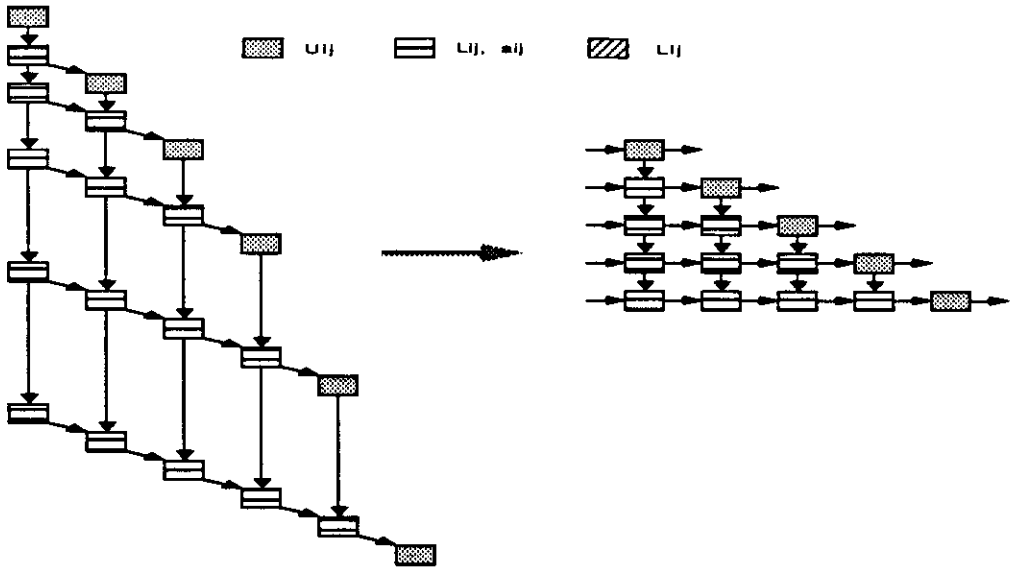


Figure 8: Reducing the number of nodes in LU–decomposition

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

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

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. In each case, we present the dependence graph of the algorithm, identify the irregularities, and describe the transformations suggested for specific irregularities.

## 5.1 Bi–directional broadcasting

Recently, S.Y. Kung et al. [5] have considered the design of systolic arrays for the transitive closure and the shortest path problems, two examples of non–regular algorithms. They state that "for an algorithm to be implemented in a systolic array, the dependence arcs in the associated dependence graph of the algorithm should have a certain regularity." In their analysis of the transitive closure and shortest path algorithms, they have found that the node which is the source of broadcasting changes its relative location within the graph and that the broadcasted data propagates in two opposite directions. They claim that these irregularities cause problems in the design, although they do not identify those problems. The broadcasting irregularity they have encountered is not only bi–directional, but bi–dimensional as well. Their methodology performs a linear mapping (i.e., a *projection*) to reduce the number of nodes in the dependence graph of the algorithm. Such mapping is affected by those algorithm irregularities. It should be noted that Kung et al. have encountered

the irregularities in the algorithm expressed in a single assignment form (i.e., a transformed version of the algorithm in which broadcasting has been replaced by data pipelining) and not in the original form of the algorithm.

Broadcasting is not desirable from the implementation point of view, since it implies global communications. Therefore, implementations normally replace broadcasting with data pipelining. That is, the broadcasted data flows in a pipelined manner through the cells of the array, reaching all the destination cells along the way. Kung et al. [5] refer to this as *transmittent data*. Notice that this is not the only possible way to implement broadcasting, since an alternative is to use replication of data.

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

### 5.1.1 Transformation of graphs having bi–directional broadcasting with independent paths

There are two issues of interest in bi–directional broadcasting, namely the synchronization of data flowing in two opposite directions and the influence of bi–directional broadcasting in the reduction of nodes in the graph. Both are complicated by bi–directional broadcasting, as we show in Figure 9. In Figure 9a, there is bi–directional broadcasting from the nodes located along the main diagonal of the graph. Synchronizing this graph requires the addition of delay registers along the vertical paths to the left of the source of broadcasting, as shown in Figure 9b. 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., sub–paths in the graph do not have the same length). The application of the procedure proposed in Section 3 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 if 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 9a. 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 9c.

The transformation outlined above allows to deal with dependence graphs that exhibit bi–directional broadcasting, but not bi–directional and bi–dimensional data flow simultaneously. We are currently looking into systematic transformations for graphs that exhibit both properties.

In the next sub–section, we present the algorithm for LU–decomposition with pivoting which exhibits bi–directional broadcasting.

### 5.1.2 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

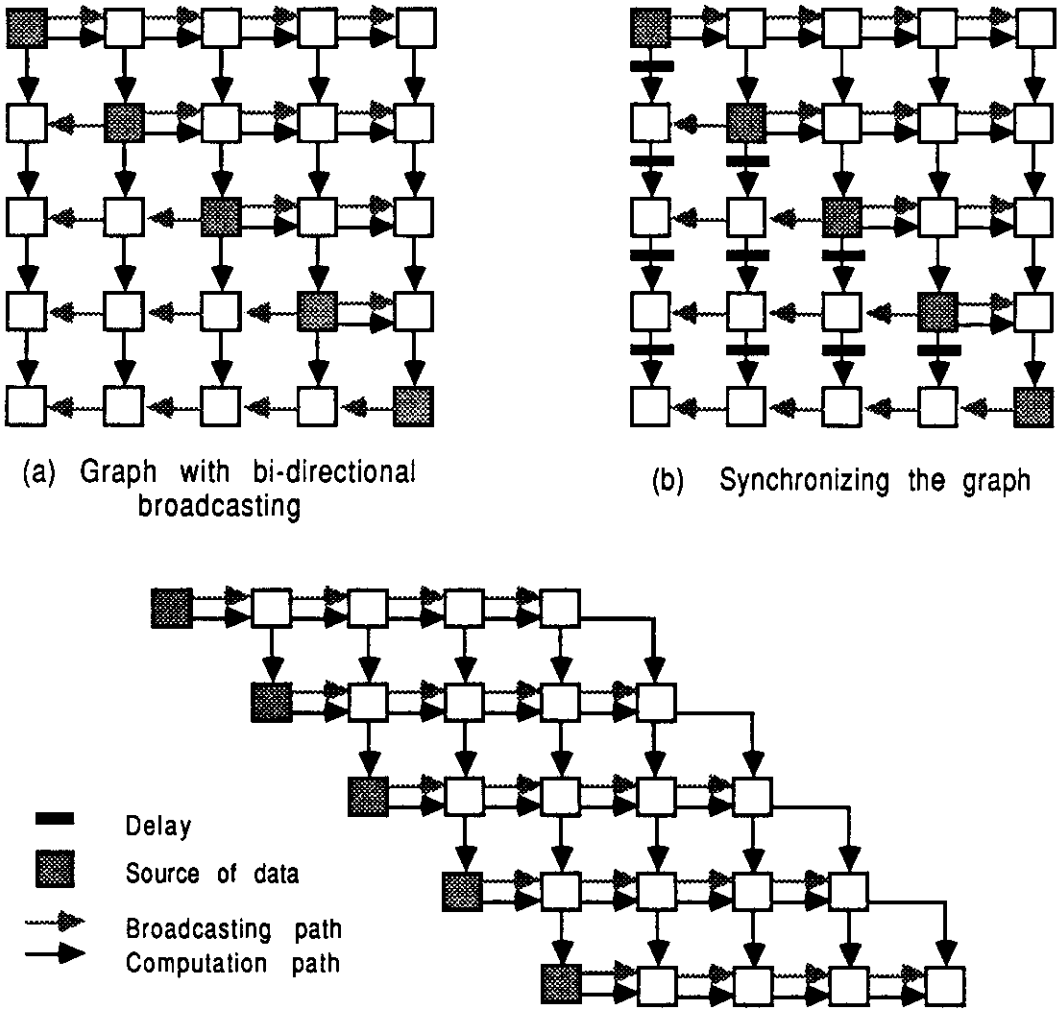



Figure 9: Independent computation paths with bi–directional broadcasting

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 [8] 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 computations. We make no specific statements regarding the suitability of this scheme from the numerical point of view.

The LU–decomposition computation with neighbor pivoting is described by the dependence graph shown in Figure 10. 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. This graph exhibits broadcasting of intermediate results and varying parallelism 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: to compute the remaining elements $u_{i,j}$ $(j > i)$, and to exchange previously computed elements $l_{i,j}$ $(i < j)$, if necessary.

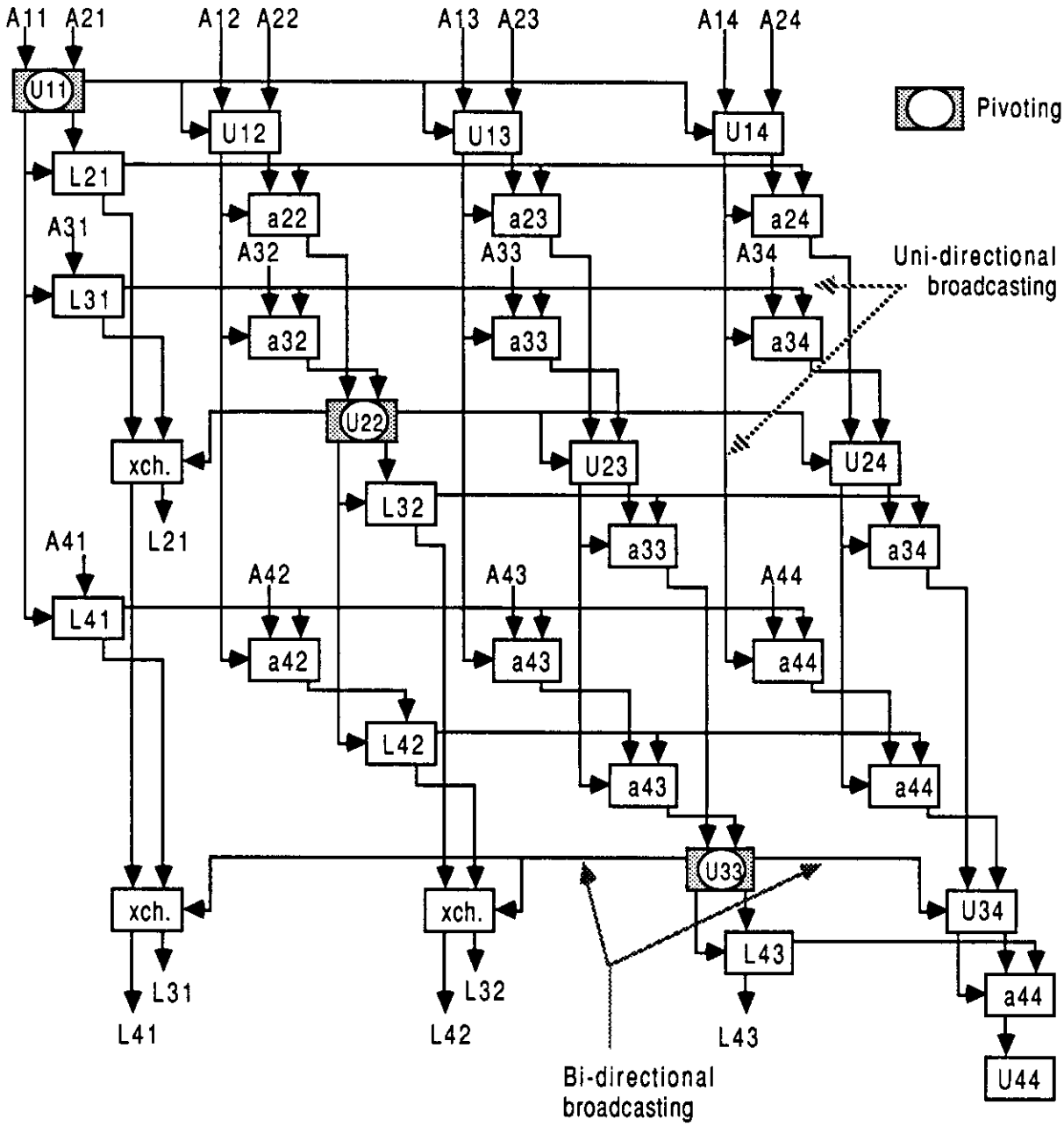
In a similar manner to the case without pivoting, we first remove broadcasting from the graph and replace it by data pipelining. The resulting graph is shown in Figure 11. Notice that the arrival of data to the nodes in this graph is not synchronized. Next, we apply transformations to this graph to deal with bi–directional broadcasting.

### 5.1.3 Bi–directional broadcasting in LU–decomposition with pivoting

In Figure 11, 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 in section 5.1.1, 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 12. The resulting graph has broadcasting in only one direction. However, the interconnection pattern between levels of the graph is non–uniform. Further transformations to solve this irregularity are discussed next.

## 5.2 Non–uniform interconnection pattern

Another type of irregularity in matrix algorithms occurs when two levels in the dependence graph have non–uniform interconnection pattern. That is, nodes at two levels are interconnected with a regular pattern excepting some interconnections at the boundaries of the level which are not the same. An example of this kind of irregularity is shown in Figure 13a. Non–uniform interconnection pattern between levels of the dependence graph complicates the collapsing of sets of nodes into single nodes, because sub–paths in the graph do not have the same structure.




Aij - input matrix
aij - updated matrix

Figure 10: LU–decomposition with neighbor pivoting

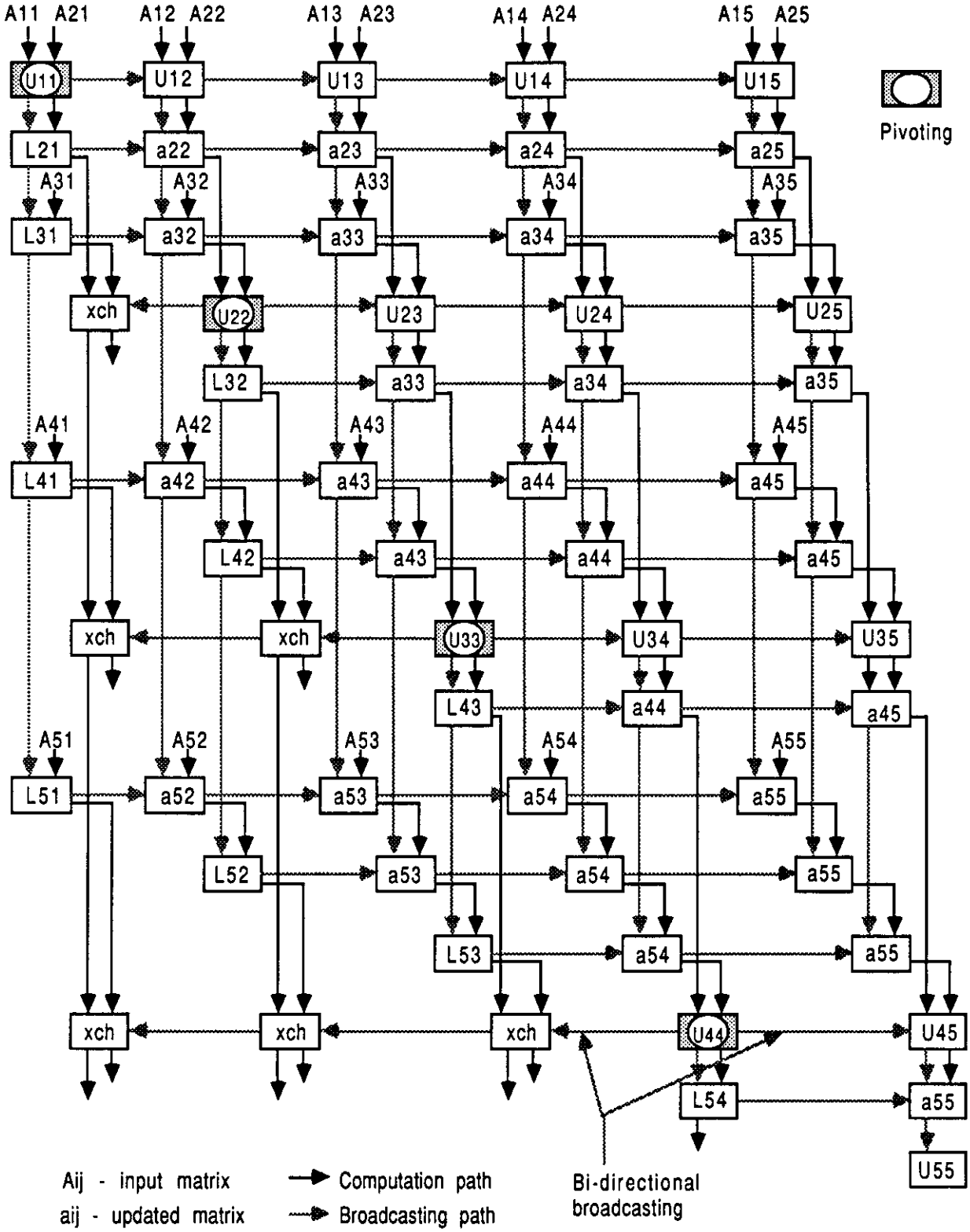



Figure 11: LU–decomposition with neighbor pivoting and no broadcasting

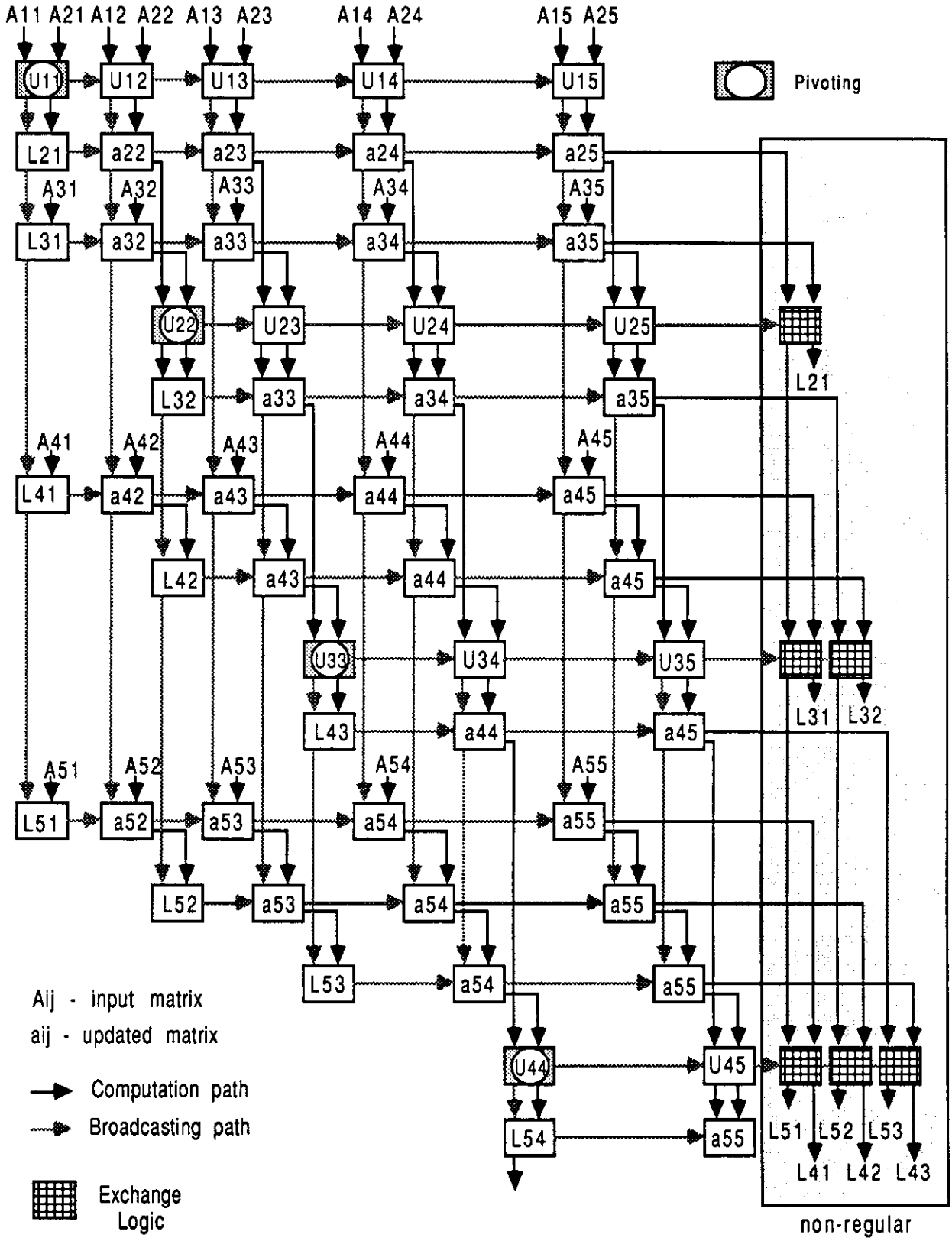



Figure 12: Transforming bi–directional broadcasting in LU–decomposition with neighbor pivoting

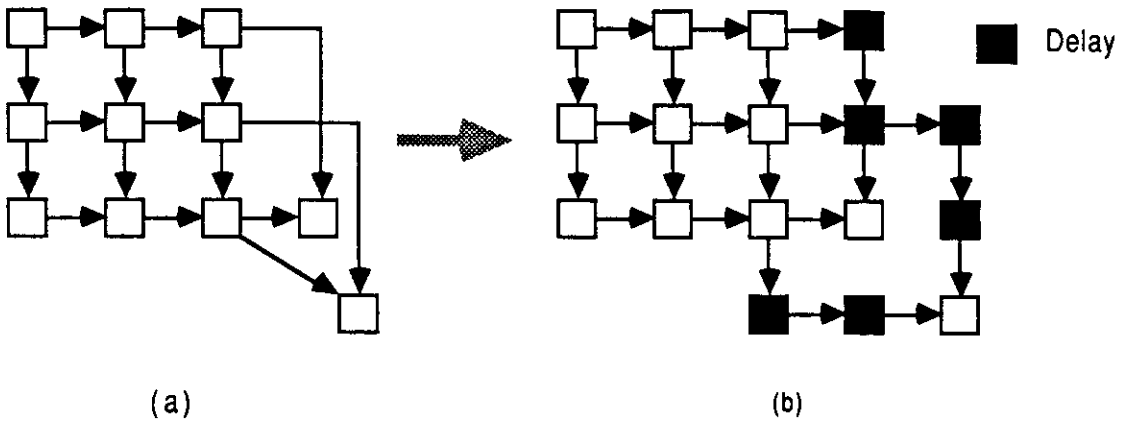



Figure 13: Non–uniform interconnection pattern

### 5.2.1 Transforming non–uniform interconnection pattern

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 at the corresponding levels. An example of this situation is depicted in Figure 13b. 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 multiple–instance computations, this increase in delay is not significant.

We apply next this transformation to the non–regular interconnection pattern existing in the transformed graph for LU–decomposition with pivoting shown in Figure 12.

### 5.2.2 Non–uniform interconnection pattern in LU–decomposition with pivoting

In Figure 12, 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 3 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 $l_{i,j}$. On the other hand, 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. Moreover, 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

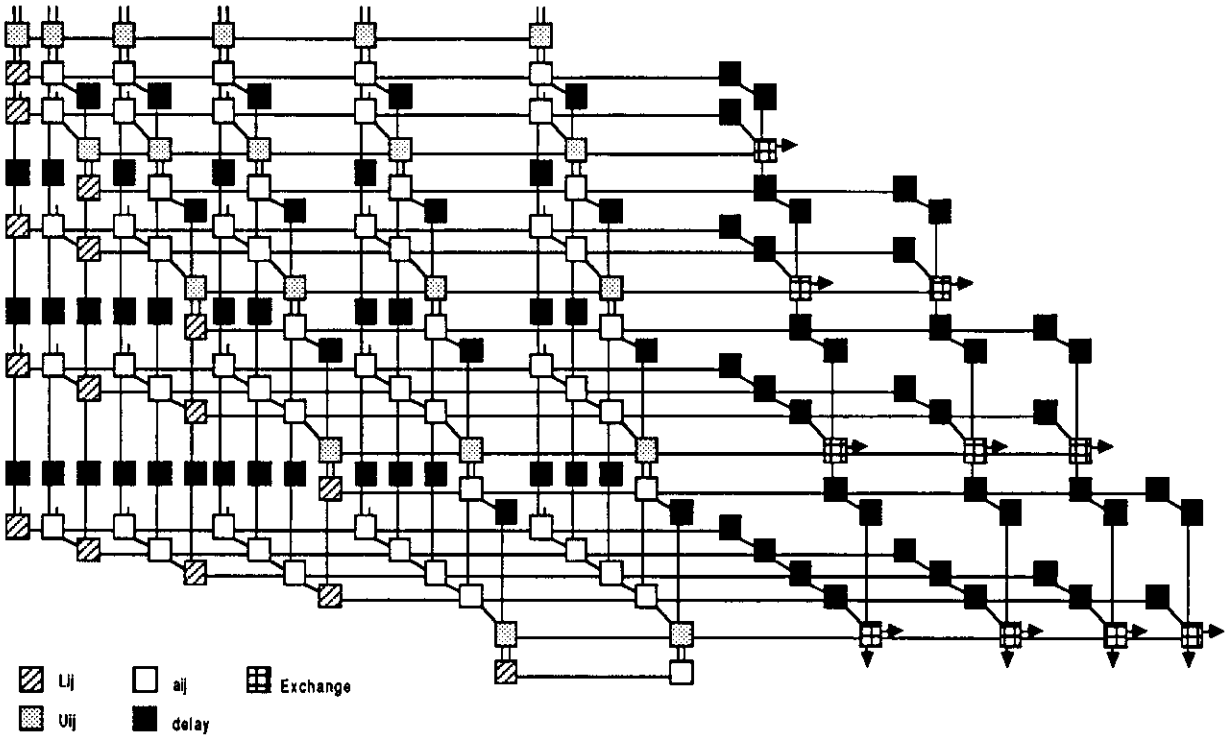



Figure 14: Regular interconnections in LU–decomposition with neighbor pivoting

the same problems just mentioned.

We apply now the transformation to deal with non–regular interconnections proposed above. 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 14. 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 is regular, therefore suitable for grouping nodes. We apply now the procedure proposed for such task in Section 3. The resulting graph, shown in Figure 15, is regular and can be mapped directly into a triangular array.

# 6 Conclusions

We have addressed the existence of some irregularities in matrix algorithms, as part of a design methodology for arrays of PEs that we are researching. This methodology consists of the systematic application of transformations on a fully–parallel graph describing the algorithm, to fulfill restrictions required for an implementation. We have identified some of those irregularities and proposed transformations to the dependence graph to remove them. The graphs obtained as a result of the proposed method are suitable for further transformations aimed towards regular algorithms, or for direct implementations as arrays of PEs. In particular, they are suitable for a procedure which reduces the number of nodes in the graph. The irregularities considered here are bi–directional

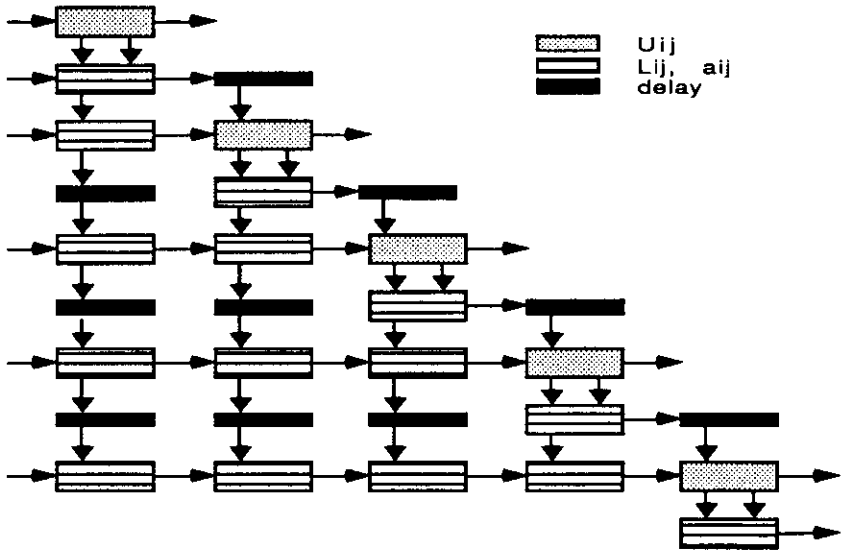



Figure 15: Regular (triangular) array for LU–decomposition with neighbor pivoting

broadcasting and non–regular interconnection pattern between levels of the dependence graph.

We have used the LU–decomposition, without and with pivoting, as example of application of our methodology to non–regular algorithms. Such application has resulted in triangular arrays for this computation, with better utilization than square arrays formerly proposed for it.

We have described the features of a few basic transformations. Their application has allowed us to show that it is possible to incorporate implementation restrictions and those irregularities as part of a systematic design method. However, these transformations are not an exhaustive collection for all possible irregularities and matrix computations. In addition, it seems that there are cases where transformations specific to a given algorithm are needed. The objectives of the 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 systematically 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] 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.

[3] 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.

[4] J. Moreno and T. Lang, "Design of special–purpose arrays for matrix computations. Preliminary results," *To be published in SPIE Real–Time Signal Processing X*, 1987.

[5] 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.

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

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

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