A PROPOSAL FOR THE SYSTEMATIC DESIGN OF ARRAYS FOR MATRIX COMPUTATIONS

Jaime H. Moreno

May 1987
CSD-870019

A Proposal for the Systematic Design of Arrays for Matrix Computations

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

Report No. CSD–870019*
May 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 propose to develop a general and systematic methodology for the design of matrix solvers, based on the dependence graph of the algorithms. A fully-parallel graph is transformed to incorporate issues such as data broadcasting and synchronization, interconnection structure, I/O bandwidth, number and utilization of PEs, throughput, delay, and the capability to solve problems larger than the size of the array. The objective is to devise a methodology which handles and relates features of the algorithm and the implementation, in a unified manner. This methodology assists a designer in selecting transformations to an algorithm from a set of feasible ones, and in evaluating the resulting implementations.

This research is motivated by the lack of an adequate design methodology for matrix computations. Standard structures (systolic arrays) have been used for these implementations, but they might be non-optimal for a particular algorithm. Reported systems have used ad-hoc design approaches. Some design methodologies have been proposed, but they do not address many important issues.

A preliminary version of the proposed methodology has been applied to algorithms for matrix multiplication and LU-decomposition. The approach produces structures which correspond to proposed systolic arrays for these computations, as well as structures which exhibit better efficiency than those arrays. The results show that different transformations on a graph may lead to entirely different computing structures. The selection of an adequate transformation is thus directed by the specific restrictions and performance objectives imposed on the implementation. The designer can identify and manipulate the parameters that are more relevant to a given application.

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. Ad–hoc design approaches have been applied in the systems that have been reported. 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 have previously devised an algorithmic model and a methodology to evaluate the effectiveness of replication, pipelining and local parallelism in the implementation of multiple–instance algorithms [3]. Such method was used to analyze the Singular Value Decomposition computation, resulting in implementations with significant improvement in efficiency with respect to the linear systolic arrays proposed for it [4].

In this research, we propose to develop 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. This methodology should provide 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.

The existence of such methodology would allow the designer to identify and manipulate the parameters that are more relevant to a given application. For example, if I/O bandwidth is critical in a certain implementation, the methodology should make it possible to take such requirement into account. In the same way, the methodology should not be restricted to use a particular architecture or interconnection structure.

We propose a methodology based on the dependence graph of algorithms. 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. The specific transformations depend on the particular parameters of interest. The proposed methodology addresses multi–instance and single–instance computations. To achieve these objectives, some transformations exploit pipelining of data to enhance concurrency and reduce communication requirements, while other transformations are oriented to reduce the computation time.

We suggest to use a fully–parallel dependence graph as the description tool because such notation exhibits the intrinsic features of an algorithm. These graphs are characterized by having all inputs and outputs available in parallel, and no loops (i.e., loops are unfolded). From such dependence graph it is possible 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. 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 or expensive interconnection structure and I/O bandwidth, and large number of units. The suggested methodology deals with these problems, while still attempting to preserve the features inherent in the dependence graph.

We have applied a preliminary version of this methodology to the algorithms for matrix multiplication and LU–decomposition. We use transformations which incorporate a subset of the design issues listed above. Although some of these transformations seem to be of general application, others seem appropriate only for specific cases. The proposed research is oriented towards identifying and providing a formal definition of a general set of transformations.

In the next section, we define the specific problem that this proposed research addresses. In section 3, we review previously proposed methodologies for the design of arrays, pointing out their major benefits and disadvantages. Section 4 describes the basic transformations in a preliminary version of our systematic methodology, and illustrate the use of these transformations by applying them to the algorithms for matrix multiplication and LU–decomposition.

2 Scope of the Proposed Research

The problem of devising a systematic design methodology for arrays of processing elements has been studied by several researchers, but it remains unsolved in many aspects [2]. As the problem is quite large and complex, it is necessary to focus on a subset of important issues, and address those issues specifically. We indicate now the scope of the proposed research.

2.1 Areas of Application of the Methodology and Admissible Algorithms

In a review of parallel processing algorithms and architectures for real–time signal processing, Speiser and Whitehouse [5] have shown that the major computational requirements for many important real–time signal processing tasks can be reduced to a common set of basic matrix operations. For example, critical signal processing tasks include adaptive filtering, data compression, beamforming, and cross–ambiguity calculation. For these applications, the basic set of required matrix algorithms includes matrix–vector multiplication, matrix–matrix multiplication and addition, matrix inversion, solution of linear systems, eigensystems solution, matrix decompositions (LU–decomposition, QR–decomposition, singular value decomposition). Since these matrix operations provide a large portion of the burden for real–time signal processing, such burden has limited the adoption or even the comprehensive evaluation of new signal–processing algorithms, permitting them to be applied only to small problems in off-line computations, or to limited data sets [5].

Thus, this research is oriented to fulfill needs in the area of high–speed implementations of matrix computations, for fields such as signal and image processing, pattern recognition, and control systems.

The objective of this research is to devise a systematic design methodology for arrays of PEs for *matrix computations*. We have selected matrix algorithms for the following reasons:

i) Matrix algorithms are compute–bound, requiring concurrent implementations to achieve high computation rates.

ii) Matrix algorithms can be decomposed into regular subcomputations, thus they are suitable for implementation as a collection of regularly connected PEs.

iii) Matrix algorithms scale regularly, so that implementations can be devised for small matrices and the results extended to larger ones.

iv) Real–time implementation of matrix algorithms are highly desirable, as discussed in section 2.1.

2.2 Design Methodology Based on Graph Transformations

In the proposed methodology, the original representation of the algorithm (a graph) is *transformed* into a form suitable for implementation (the properties of the graph are discussed later). Thus, our methodology follows a transformational approach, the same as other previously proposed schemes. Transformations are guided by implementation restrictions which render the original graph not feasible as an implementation. The objective is to provide a unified treatment of algorithm features and implementation restrictions.

Our method deals explicitly with the restrictions existing for a given implementation. Those restrictions include I/O bandwidth, utilization of PEs, limited data broadcasting, interconnection and communication structure, number of operation units. The designer must select from such restrictions those which are relevant for a particular implementation, resulting in different alternatives depending on the restrictions selected and the order in which those restrictions are taken into account. This is in contrast to other proposals, where emphasis is placed mostly on interconnection regularity and strictly nearest–neighbor communications. The remaining implementation issues are largely ignored during the application of those methodologies, although for a given algorithm such issues might be as important as the ones considered.

2.3 Number of Instances of the Computation

An important parameter that influences a suitable implementation is the *number of times* that an algorithm is executed on independent data. Pipelined implementations are effective only in cases where this number is large with respect to the number of stages in the system. On the other hand, if the algorithm is computed just once, only parallelism is effective to reduce the computation time. Thus, we must distinguish between implementations for multi–instance and for single–instance algorithms, and deal with them differently.

The proposed methodology addresses multi–instance and single–instance computations. Some of the transformations exploit pipelining of input data and of intermediate results, to enhance concurrency and reduce communication requirements. In these cases, the objective is to increase throughput. Thus, the target of those transformations are multi–instance algorithms, or computations which can be decomposed into multiple instances of basic algorithms. Other transformations are oriented to reduce the computation time of the algorithm, and are appropriate for a single evaluation of the computation. The selection of suitable transformations depends on this degree of multiplicity.

2.4 Degree of Automation in the Design Process

For a transformational methodology, an important issue is the degree of automation desired in the process. Many methodologies proposed in the literature state as their goal to completely automate the design process. However, selecting a good transformation at any step in the process is not an obvious task, unless the number of choices is small and they can be evaluated exhaustively. Straight-forward transformations may fail, so that the synthesis procedure usually requires creativity from the designer.

The proposed design methodology is oriented towards assisting the human designer, providing him/her with a flexible tool able to incorporate the design requirements. The design process is an iterative, perhaps interactive, procedure in which the designer selects a transformation from a set of feasible ones, applies it, and evaluates the result according to some performance and cost measures defined for the implementation. In this way, the design becomes a search in the space of solutions available through the transformations, for the alternative which offers the best cost–performance trade–offs. This search is assisted by the methodology.

2.5 Model of Computation for the Implementation

Our methodology uses a synchronous model of computation for the implementation, that is, all cells in the array operate synchronously, and all data transfers are performed simultaneously. There are no pre–defined restrictions regarding interconnection and communication structure, or I/O data bandwidth. Thus, the methodology is not restricted to a particular architecture.

2.6 Criteria for Optimality and Evaluation of Implementations

Criteria for optimality and evaluation of implementations are controversial, since the definition of optimality may not be the same for every implementation. The design of VLSI circuits has used A, T, AT and AT^2 as measurements of efficiency (where A is area and T is time) [6],[7]. These measures have been used as "lower bounds to solve problems in the VLSI domain" [6].

If an array of processing elements was to be fabricated as a single VLSI device, then the use of these measures would have the same value as for any other VLSI design. However, it is usually the case that implementations of arrays with current technology do not fit into a single VLSI circuit for most practical examples. Instead, these implementations are arrays of devices, with one or several components per PE [8], [9], [10]. In this context, area–time tradeoffs are no longer as meaningful as in the VLSI domain. Furthermore, area might be a complex function of the number of PEs, number of buffers, interconnection pattern, etc. [11]. Instead, significant parameters may be number of devices (or PEs), throughput, I/O bandwidth, or others. In spite of these facts, the measures mentioned above have been used to evaluate systolic structures [11], [12], where area has been computed as number of PEs. Note, though, that the advent of wafer scale integration (WSI) may bring the former measures back into adequate use.

We suggest to avoid pre–defined criteria for optimality. Our approach to this issue is to provide in the methodology enough flexibility so the designer can choose the measures of interest for the implementation, and evaluate such implementation according to those measures. In this manner, the methodology exhibits adaptability for different technologies as they evolve in time.

Thus, we propose to devise a methodology in which measures of efficiency and optimality in the design are selected by the designer, from a predefined set, tailoring these parameters to the implementation at hand. This set of measures includes delay (i.e., computation time), throughput, number and utilization of operation units, speed–up, efficiency (i.e., speed–up over number of units). The methodology should provide a mechanism to define such measures, and procedures to evaluate with respect to those measures different designs resulting from the methodology.

2.7 Description of the Algorithms

The description of an algorithm is very important, since the outcome of the methodology depends on it. Possible description tools are mathematical expressions, program with loops, program in a parallel high–level language, graphical description (we discuss later the impact of the description of the algorithm on the methodologies proposed in the literature). Our approach uses a graphical description of the algorithm, as indicated below.

Fully–Parallel Graph as Description Tool

We have selected a fully–parallel dependence graph to describe the algorithms. In such graph, nodes represent the operations and edges correspond to data communications. All inputs are assumed available simultaneously, and all outputs are available as soon as they are computed. Loops are unfolded completely, and branches are expressed explicitly.

The graph doesn't include synchronization of the arrival of data to the nodes. Thus, nodes fire when their operands become available, same as data–flow graphs. The major difference between data–flow graphs and fully–parallel graphs is the absence of "tokens" in the latter. These tokens are not needed, since the model of computation we use for implementation is synchronous, and the arrival of data to the nodes is synchronized as the result of graph transformations during the application of the methodology.

We select to use a fully–parallel dependence graph to describe the algorithms, because such notation exhibits the intrinsic features of an algorithm. From the dependence graph it is possible 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. Figure 1 shows an example of this approach. The resulting structure, which is a *pipelined implementation of the graph,* has the following features:

- minimum delay (determined by the longest path in the graph)
- it provides a new result every cycle (throughput $T = 1[evals/cycle]$)
- optimal utilization of PEs when computing multiple instances of the algorithm

However, the pipelined implementation of the graph may require complex or expensive interconnection structure and/or I/O bandwidth, and large number of units. The proposed methodology deals with these problems, while still attempting to preserve the features inherent in the dependence graph.

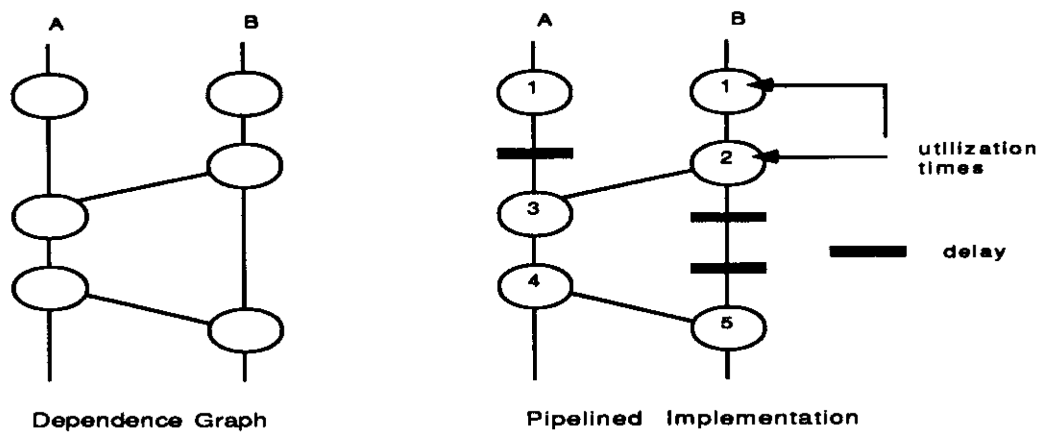

Figure 1: A dependence graph and its pipelined implementation

2.8 Mapping of Problems Larger than Array Size

Computational arrays are usually characterized by the size of the array, which defines the size of the problems that can be solved in such arrays. Many applications often require the processing of large matrices, but it is not feasible to build an array as large as the dimension of such matrices. A similar situation arises when the same array is used to solve problems of different size. In these cases, it becomes necessary to decompose the problem into subproblems, so that the subproblems fit into the target array. This is known as the *partitioning problem.*

We expect to include the partitioning problem in our methodology, since we envision partitioning as an integral part of it. The dependence graph of an algorithm is drawn for the size of the problem, not for the size of the target array. In this way, all data dependences are available to the designer to manipulate them in the most convenient manner.

2.9 Formal Definition of Transformations

Since the proposed methodology is a transformational one, it is necessary to identify a general set of transformations. In addition, it is extremely important that the transformations are proven correct, since the methodology relies on them to achieve its objectives. Thus, it must be proved that for any algorithm the application of a transformation on a dependence graph preserves the function which is expressed by such graph. If such proof exists, the implementations resulting from the application of the methodology will not need to be verified. An implementation is obtained as the result of applying correctness–preserving transformations, and the sequence of transformations itself serves as a constructive proof of correctness of the implementation.

As a consequence, a formal definition and proof of correctness of each transformation is necessary. We expect to achieve this formalism using some symbolic representation of the transformations, so that the same symbolism may be used to prove correctness. Methodologies which use mathematical expressions as the description tool have this property already included. Those that follow other transformational approaches have also had to deal with this issue. Note that we advocate the use of the formalism only to prove the correctness of the transformation, and not as part of the methodology.

In [13], Sheeran uses graphical descriptions to illustrate some transformations formally defined in μFP, an algebraic VLSI design language based on functional programming. We envision an approach in the opposite direction, namely each transformation on the graph has an algebraic counterpart that proves it correct.

We have presented the scope of our proposed research, indicating our approach to face the problem. In the next section, we review previously proposed methodologies for the design of arrays of PEs, and look into how those proposals relate with our approach to the problem.

3 Previously Proposed Methodologies for the Design of Arrays

The development of a systematic methodology 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, and new ones have been suggested since then. Fortes et al. conclude that the most common characteristic of the proposed methods is the use of a *transformational approach*, i.e., "systolic architectures are derived by transforming the original algorithm descriptions that are unsuitable for direct implementation." In other words, the proposed systematic design methods consist of transformations of a high-level specification of a problem 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, lower-bound on the utilization of processing elements, or limited data broadcasting. Furthermore, the success in achieving their goal is not convincing as suggested by Fortes et al., who state that "from a global point of view, it is clearly indicated that the two largest limitations in the state of the art of existing transformational systems are the non-existence of powerful systematic semantic transformations and the inability to systematically achieve optimality in the resulting designs" [2].

In this section, we review previously proposed methodologies for the design of arrays. First, we discuss some classifications of those methodologies, and later look into the impact of the algorithm description in the resulting implementations.

3.1 Classifications of Methodologies

There are several alternatives to classify methodologies for the design of arrays of PEs. Fortes et al. [2] characterize existing transformational approaches as follows:

i) How algorithms are specified, that is, what mechanism or tool is used to present the algorithm to the transformational methodology. This can be a high-level language description, pseudo-code, mathematical expressions, or even a natural language description.

ii) What formal models are used, that is, what representation is used to abstract the relevant features of an algorithm. This model can be based on the functional semantics of an algorithm (i.e., algebraic expressions), on the structure of an algorithm (i.e., a graph), or a geometric description of the algorithm.

iii) How systolic architectures are specified, that is, what hardware model or level of design is used to describe the array.

iv) What types of transformations are used on and between the representations. That is, whether transformations are systematic or ad–hoc, and whether they are used to go from one type of representation to another, or to the same type of representation but at a different level.

Emphasizing on how and at which level the proposed transformations are applied, Li and Wah [11] classify the methodologies as follows:

1. Transformations performed at algorithm representation level, and direct mapping made from this level to architecture.
2. Transformations performed at algorithm–model level (i.e., at the level corresponding to item (ii) above), with procedures for deriving the model from the algorithm representation and for mapping the model into hardware.
3. Transformations performed on a previously designed architecture to obtain a new architecture.
4. Transformations performed to map a systolic architecture into the function implemented, and to prove the correctness of the design.

From these classifications, we can recognize the importance of the description of an algorithm. Description tools being used are:

- mathematical expressions
- graphical descriptions
- loops and begin–end blocks
- programs in high level language

We look now into how these description mechanisms have been used on the different methodologies. We can anticipate that although some transformations available with those descriptions may be powerful, the resulting schemes are suitable only for the classes of algorithms which have representations as required by the corresponding methodology (i.e., uniform recurrence equations, canonical algebraic expressions, recursion equations, program with loops). Furthermore, it is not always clear whether adequate transformations can always be found with those descriptions, and the target architectures may be restricted as a result of those tools.

3.2 Algorithm Descriptions

We focus our discussion on the impact that the description of the algorithm has on the results obtained with the proposed methodologies. In most cases, the objectives pursued by the transformations are the same, namely to eliminate data broadcasting, and enhance local communications and regularity. Only the form of the transformations is different, since the representations are different.

Our methodology is not too distinct in this respect, since we also apply transformations to a description of an algorithm (a graph), with similar objectives. Consequently, some of the underlying ideas in the methodologies discussed here are suitable to our approach. The main difference relies in our attempt to incorporate most implementation restrictions, if not all, as part of the design process. This is in contrast to the other schemes, where implementation issues such as I/O bandwidth, for instance, are a result of the application of the methodology, with no control over it.

Methodologies that require the algorithms to be described in a particular manner are limited to those algorithms that have such a description. That is the case, for instance, of those schemes which use recurrent expressions. Since our approach is more general than that, requiring only to draw the dependence graph of the algorithm, it might be the case that for certain specific classes of algorithms other methodologies are more convenient. We have no definite answer regarding this issue yet. Thus, our discussion below about previously proposed methodologies doesn't address this topic.

Mathematical Expressions

The manipulation of mathematical expressions to derive arrays of PEs allows to apply powerful algebraic transformations to an algorithm. These transformations attempt to enhance pipelining and local communications by index transformations, that is adding indices to existing variables, renaming variables, or introducing new variables. The resulting expressions are in some "canonical" form, which corresponds to structured sets of computations written as recurrence relations or nested loops [14].

Kung and Lin, for example, have proposed an algebra for systolic computation [15]. Algebraic transformations are applied to the algebraic representation of an algorithm to obtain an algebraic representation of a systolic design. They use this algebra to derive designs which have what they call the "systolic property," that is, designs where broadcasting has been replaced by distributing common data to different destinations at different times. Such design is represented as a Z–graph, where there is an edge for each variable and a node for each computation. Edges have labels of the form z^{-k}, where k denotes the delay between nodes. The Z–graph representation is readily mappable to hardware.

Guerra and Melhem [14] address the design of systolic systems with non–uniform data flow, based on a subset of the data dependences extracted from the original problem specification. Their approach identifies chains of dependent computations which are converted into recurrence equations, and then maps the new specification into hardware.

Li and Wah [11] formulate the design as an optimization problem, where the search space is polynomial on the problem size. They propose a methodology for the design of optimal pure planar systolic arrays for algorithms that are representable as linear recurrence processes, using completion time T or area–time AT^2 as figures of merit for the design. Their systolic designs are characterized by three parameters, namely velocities of data flows, spatial distribution of data, and periods of computation. They express completion time and hardware complexity in terms of these parameters.

Quinton [16] uses a set of uniform recurrent equations over a convex set D of cartesian coordinates as the description of the algorithm. The proposed methodology first finds a timing function compatible with the dependences of the equations, and then maps the domain D into another finite set of coordinates. However, it is not clear how the mapping is systematically performed. Furthermore, the method is limited to cases where one of the data streams is dependent on other data streams [17].

Weiser and Davis [18] propose a method for treating sets of data as wavefronts entities, and apply transformations to the wavefronts. The problem is presented as operations on sets of data, using a KM function on such data. Thus, wavefronts, KM functions and the transformations need to be identified. The method is best applicable to algorithms that can be described by relatively simple and concise mathematical expressions [2].

The nature of the mathematical expressions used in these methods may lead to inadequate conclusions regarding the features of an algorithm. A significant example of this situation is found in the pioneering work of Kung [19], where he concluded that "LU–decomposition, transitive closure, and matrix multiplication are all defined by recurrences of the "same" type. Thus, it is not coincidental that they can be solved by similar algorithms using hexagonal arrays." A similar statement is found in [20]. However, matrix multiplication and LU–decomposition have quite different dependence structures, so that they are not mapped efficiently into similar arrays [21].

Furthermore, schemes using mathematical expressions are suitable only for the classes of algorithms which have representations as required by the corresponding methodology (i.e., uniform recurrence equations, canonical algebraic representations, recursion equations).

Loops

Program with loops as the starting point of a methodology have been used in [20] and [22]. In [20], Moldovan first "transforms the algorithm with loops into a highly–parallel form suitable for VLSI, and then transform the resulting algorithm into a systolic array." The idea is that the data dependences of the new structure can be selected in the transformation process. Unfortunately, this work does not describe a systematic way to find the transformation [16]. Moreover, it is suited only to the class of algorithms which can be described by programs with loops (or recurrence equations) [2].

Miranker and Winkler's work [22] is similar to Moldovan's approach. Extensions are the rewriting of expressions by using properties of the operators in an ad–hoc manner, and the use of graph embeddings based on the longest path of a computation graph when such graph is too irregular. Theoretically, this approach applies to any algorithm, although systematic design seems possible only for those algorithms described by programs with loops [2].

Moldovan's work brings up an important issue which has also been recognized as such by other researchers: the data dependences are the clues to potential transformations [17]. Moldovan's approach uses this fact to perform linear transformations on those dependences, with the main goal of eliminating data broadcasting and global communications. Our approach also uses the data dependences, although we do not require the algorithm to be described in terms of loops.

Graphical Descriptions

Since data dependences in an algorithm contain relevant information required for an implementation, it seems appropriate to use those dependences as the description of the algorithm, and the starting point of a design methodology. We discuss now some approaches in this direction.

Ramakrishnan et al. [23] proposed a formal model for a linear array of processing elements, and graph representations of programs suitable for execution on such a model. These graphs were defined as homogeneous graphs, which are a more limited class of program graphs than general data–flow graphs. In particular, homogeneous graphs have the same number of edges into and out of every node, excepting those nodes representing sources or sinks of data (i.e., inputs or outputs, respectively). The proposed methodology first partitions the graph into sets of vertices that are mapped into the same processing element. In a second step, a syntactically correct mapping is used to map computation vertices onto processors and time steps, and labels and edges to communication delays and interconnections [2]. This methodology applies only to homogeneous graphs with connected subgraphs satisfying certain properties. Moreover, the method can only be used to generate linear arrays [2].

Another graphical approach has been pursued by Barnwell and Schwartz [24]. This method starts with an algorithm described as a fully–specified flow graph, that is, a directed graph in which nodes represent operations and edges represent signal paths. Nodes are also used to represent delays explicitly, when those delays are part of the algorithm (e.g., digital filters). This description tool seems to be almost the same as the fully–parallel graph we use in our approach. The major differences are the requirement on the nodes to be fundamental operations performed by the cells in the implementation, and the fact that this approach is oriented to implementations with identical cells. The latter characteristic arises as a result of targeting the methodology to implementations in multiprocessors.

Barnwell and Schwartz use two different models of computation: a synchronous model, which they associate with systolic arrays, and a "skewed single instruction multiple data" model (SSIMD), where the same program is executed in each cell in the array, and that program realizes exactly one time–iteration of the flow–graph. In other words, they exploit the multi–instance property of algorithms to allocate a separate processor to each instance of the computation. We have provided a more formal characterization of this situation when comparing implementations using replication and pipelining for multiple–instance algorithms [3].

Barnwell and Schwartz claim that since systolic arrays are characterized by synchronous data transfers, flow–graphs are constrained to have every output from a cell terminated by a delay node (or pipeline register). Hence, "the generation of systolic solutions for flow–graphs reduces to the distribution of delays nodes throughout the flow–graph." Their methodology consists of systematic manipulation of the flow–graphs into systolic forms, using theorems from graph theory. Although their assertion is true, the methodology is not as general as we claim necessary since it doesn't deal explicitly with other important issues such as input/output bandwidth, limited data broadcasting, or lower bound on efficiency. Furthermore, their transformations are ad–hoc instead of systematic [2].

Jover and Kailath [25] proposed a pseudo–graphical approach. They introduced the concept of *lines of computation LOCs*, which are useful to determine whether a given topology is suitable for systolic computations. *LOCs* are a summary of an architecture in time or space, and some properties of the architecture can be inferred from such *LOCs*. Jover and Kailath's work includes the definition of ***systolic-type arrays***, a generalization of systolic arrays where there may be different cells not only at the boundary of the array, but also inside. This idea deserves to be considered, since implementations may benefit from not being restricted to identical cells. However, *LOCs* are not general since they require the computation of the algorithm to be distributed through the array (i.e., they do not allow a result to be "accumulated" locally in a cell). Moreover, *LOCs* are not really a design methodology.

Approaches Using High-Level Languages

General purpose high-level languages have also been used as the starting point of design methodologies. Lam and Mostow [26] use a high-level algorithm description language, and model the design process as a series of transformations on this description. They rely on the human designer to decide which transformation to apply, instead of aiming towards a fully automated approach. A computer-aided transformational tool is used to assist the designer in the process. The design process first performs software transformations on the description of the algorithm to prepare it for systolic implementation. This initial transformation converts the algorithm into a representation composed of highly repetitive computations, expressed in terms of nested-loops and begin-end blocks. This step includes the annotation of the algorithm description with statements to indicate how subfunctions should be evaluated, such as "in parallel" or "in place" (i.e., sequentially within the same unit). The automated software tool knows how to map such constructs onto hardware. Then, a sequence of hardware allocation, scheduling and optimization phases are applied iteratively. The optimization phase is guided by the user, who selects the transformations to apply [26]. The method can only process algorithms with simple loops and begin-end blocks, simple unnested function calls, scalar and array variables. It cannot deal with other high-level software constructs [2].

The underlying approach and some of the transformations used in Lam and Mostow's work is quite similar to what we propose for our research. The main difference is due to the representation of the algorithm. Lam and Mostow claim that "approaches that abstract a computation in terms of its data dependencies assume that the computation has already been decomposed into operations corresponding to the behavior of individual cells. This assumption is impractical for complex computations where the design process may depend on details of internal cell behavior." While this may be true for some of the methodologies proposed in the literature, we believe that exploiting those dependencies through a methodology which exhibits them clearly provides more elements to face the design task and achieve suitable implementations.

Chen [17] uses a general purpose parallel programming language as the description tool. An algorithm specified in such language is improved by transformations that remove broadcasting and limit the number of fan-ins and fan-outs. Another phase of transformations incorporates pipelining and attempts to fully utilize the hardware resources. These transformations are algebraic manipulations on the expressions in the parallel programming language, so that the language must be amenable to algebraic modifications. Thus, the approach is highly mathematical and therefore subject to the same limitations as described before for schemes based on mathematical expressions. Furthermore, it incorporates only the implementation issues indicated above.

Chapman et al. [27] use the OCCAM programming language for algorithm description, simulation and eventually implementation. Although the proposed approach yields programs which could be used on an array of Transputers (the INMOS processor), the objective of this work is the utilization of the OCCAM algebra to aid in the design. Chapman et al. claim that an OCCAM program can be interpreted as an algebraic description of a regular array architecture that implements a given algorithm. The OCCAM program can be transformed and, as long as the algebraic rules are adhered to, the designer can assume that the program will implement the same algorithm. However, there is no systematic way to perform those modifications, neither a mechanism to enforce only valid transformations.

3.3 The Partitioning Problem

The methodologies proposed in the literature have not explicitly considered mapping of problems larger than the size of the array. This problem has been studied extensively in other contexts, and ad–hoc solutions have been suggested in a manner similar to the design of arrays [28], [29], [30], [31]. These schemes basically assume the existence of an array, and partition the problem to map it into such an array. In the process, they may resort to transformations of the original problem so that it fits in the array (e.g., transforming a full matrix into a band matrix). These approaches require extra computational work to first decompose the problem, and then to combine the results from the component parts since they may overlap. Alternatively, such schemes may require different types of arrays to implement the sub–algorithms that compose the original problem. Our approach to the partitioning problem is different, since we intent to incorporate it as part of the design methodology.

3.4 Criteria of Optimality

Most proposed methodologies do not deal explicitly with optimality. They address some implementation issues, in particular nearest–neighbor communications and regular structure, without further evaluation of the results. In general, the methods are unable to systematically achieve optimality in the resulting design [2].

However, a few papers have addressed the topic of optimality in the design [11], [12]. For example, Li and Wah [11] proposed a methodology which measures the merit of a design in terms of the computation time (T), or the product of the VLSI area and the computation time (AT), or the product of the VLSI area and the square of the computation time (AT^2). The effectiveness of these measures has been discussed in section 2.6.

Ramakrishnan and Varman [12] present a family of linear–array matrix multiplication algorithms on a pipelined linear array. These algorithms exhibit a tradeoff between the number of processing cells and the local storage in a cell. However, the total time and storage requirements remain invariant in this tradeoff. This work provides different schemes to multiply matrices in linear arrays, all with the same area (defined as the product of the number of cells and the storage per cell), and the same computation time. However, it doesn't address the issue of optimality for other implementations of matrix multiplication, neither of other algorithms.

3.5 Basic Limitations in Previously Proposed Methodologies

From this review of previously proposed methods for the design of arrays of processing elements, we can conclude that the goal of devising a systematic methodology for this type of structures is far from being achieved. Some limitations of proposed methodologies are easily identified. Among them, we can mention the following ones:

i) Most of the proposed methodologies are oriented towards the design of standard structures (i.e., systolic arrays). However, such structures might be non–optimal for a particular algorithm.

ii) Proposed methodologies do not take into account certain implementation restrictions such as I/O bandwidth, lower bound on utilization of PEs, limited data broadcasting capabilities. They deal mainly with interconnection structure of the resulting arrays, and removal of data broadcasting.

iii) Proposed methodologies are too restrictive regarding the form and the implementation of the algorithms. That is:

- they are suitable only for algorithms which can be expressed in a given form
- they have strict implementation requirements regarding classes of PEs, fan–in/fan–out characteristics, data broadcasting

iv) Description tools used may not convey all the information about an algorithm, leading to inadequate conclusions.

v) Proposed methodologies do not have well defined criterias for optimality in the design.

vi) There is no systematic evaluation of implementation parameters.

In the next section, we present the basic features of a preliminary version of our methodology, which attempts to solve some of the current problems in this field.

4 A Graph-Oriented Design Methodology for Arrays of PEs

We present now the basic features of a preliminary version of a systematic design methodology for arrays of PEs. This is a transformational methodology, which uses a fully–parallel dependence graph as the description of the algorithm. In such graph, nodes represent the operations and edges correspond to data communications. Transformations are applied on the graph to incorporate implementation restrictions. The specific transformations depend on the particular parameters of interest at a given time. The graph used in this approach is called fully–parallel, because all inputs are assumed available simultaneously, and all outputs are available as soon as they are computed. 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.

We have applied this preliminary version of the methodology to the algorithms for matrix multiplication and LU–decomposition. We have used transformations which incorporate a subset of the issues arising in a design. Although some of these transformations seem to be of general application, others seem appropriate only for specific cases. The proposed research is oriented towards identifying and providing a formal definition of a general set of transformations.

We first describe some transformations to dependence graphs, under certain constrains of interest. Later, we will apply such transformations to specific algorithms.

4.1 Restricting I/O bandwidth

Let's assume that the I/O bandwidth of an implementation is limited, so that it is not feasible to provide the bandwidth needed by the fully-parallel dependence graph. In such case, the entire parallelism available in the algorithm can't be exploited, due to lack of data. Under these circumstances, the implementation of the graph is data–bound.

To take the I/O restriction into account, we modify the dependence graph by *introducing additional nodes to represent the input/output pads.* Since these pads are shared by different edges of the graph, we need to modify the graph so that no contention resulting from the use of the shared resource arises. To achieve this for output pads, we add delay nodes to some of the edges of the graph. For input pads, the data is delayed because it is brought into the computing structure serially through the pads. In this way, we modify the fully parallel–graph by introducing time-multiplexing of the shared resource (i.e., the pads). The application of this transformation has the following characteristics:

- The I/O restriction is applied either to input or output pads first.

  As a consequence of restricting the input pads, the requirements for output pads may decrease since all results may not be available at once. A similar situation is true for input pads if output pads are restricted.

- The assignment of edges of the graph to input/output pads is performed using the length of the paths in the graph as the selection criteria. For simplicity, we describe the methodology for output pads only. This is accomplished as follows:

  - edges belonging to shorter paths are assigned to the output pads first, expecting that longer paths can use those pads at a later time.
  - when paths have identical length, we use the knowledge about the algorithm to carry out the assignment.

Figure 2 shows this class of transformation for the output portion of a given algorithm. In this example, k groups of n outputs each have been multiplexed into a single set of n outputs. Delay nodes have been added to the edges of the graph to avoid contention for the output pads, as indicated above.

Restricting output pads implies that more than one cycle is needed to read results out of the processing array. Similarly, the restriction in input pads forces to use more than one cycle to load the data into the processing structure. Thus, throughput $T = 1[eval/cycle]$ is no longer possible. Therefore, a pipelined implementation of the modified graph will not result in optimal utilization of operation units. Additional transformations to increase such utilization will be described later.

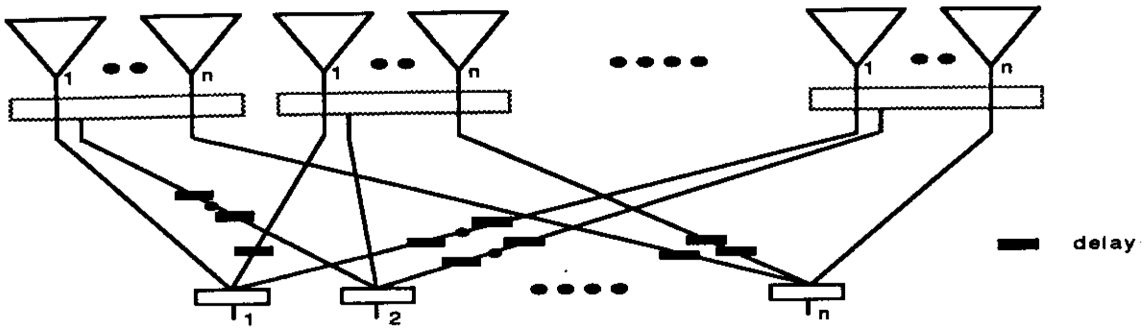

Figure 2: Restricting I/O bandwidth: output pads and delays added to matrix multiplication

4.2 Removing Data Broadcasting

We describe now a transformation to eliminate data broadcasting by replacing it with data pipelining. We apply the following methodology:

- For each broadcasted data element, identify all paths in the graph from the broadcast point to the outputs. Compute the length of each of such paths to the corresponding outputs. Add a new node to the dependence graph, representing a delay element, with its input connected to the broadcasted data. Connect all paths of broadcasted data, excepting the longest one, to the new node. Repeat this process if broadcasted signals remain present in the graph.

  Figure 3a shows this transformation. In this case, two iterations adding delay nodes were required because the topmost broadcasted signal reaches three nodes.

- After all broadcasting has been removed, synchronize the arrival of data to the nodes by adding delay nodes in the shorter paths. This can be achieved by adding the delay nodes using breadth-first search starting from the nodes associated with the data inputs. Figure 3b shows this new transformation.

- Combine computation nodes and delay nodes wherever possible. Replace delay nodes at the input by delaying the input of data itself. Figure 3c depicts this modification.

If a broadcasted data element has equally long paths through the graph, placing the delay nodes would require exhaustive search to identify which paths should not be delayed. Such selection has to be evaluated in terms of the design parameters. To reduce this problem, it is possible to perform the selection taking advantage of knowledge about the algorithm. Such a situation is illustrated in section 4.5, where this kind of transformation is applied to specific algorithms.

4.3 Utilization of PEs

As pointed out in section 4.1, a graph which has been modified by a transformation such as restricting I/O bandwidth might not have throughput $T = 1[eval/cycle]$. In such a case, a pipelined implementation of the dependence graph yields suboptimal utilization of units. It is possible to improve that utilization by *assigning several nodes of the graph to each PE.* To do so, we modify

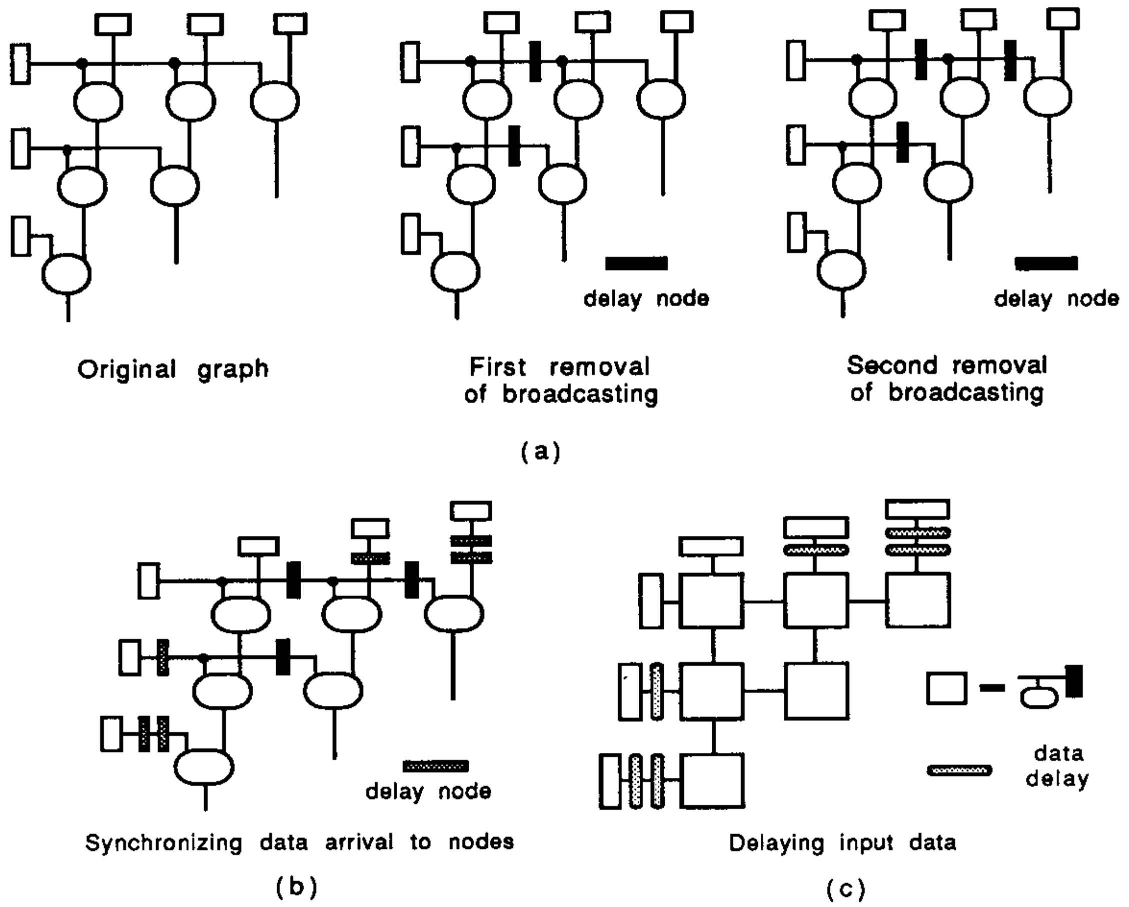
delay node
delay node
Original graph
First removal of broadcasting
Second removal of broadcasting
(a)
delay node
Synchronizing data arrival to nodes
(b)
data delay
Delaying input data
(c)

Figure 3: Removing data broadcasting

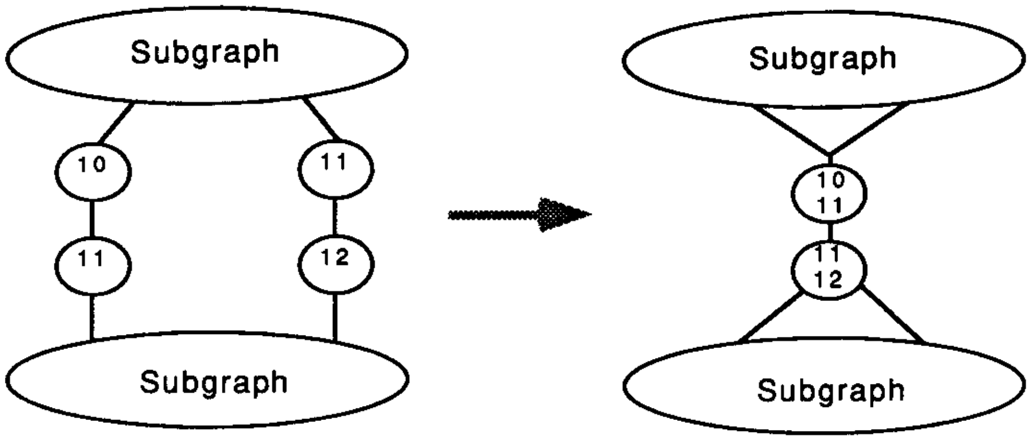

Figure 4: Utilization of PEs: collapsing identical paths

the graph by collapsing groups of nodes into a single node, and assigning the resulting node to a PE.

Our methodology carries out this transformation as follows:

- Annotate each node with the time at which it is used. This information is obtained by traversing the graph from input to output. We assume the same computation time for all nodes.
- Collapse identical paths in the graph which have different utilization time.
- Serialize onto the resulting input path the data used for the different paths which have been collapsed.
- Assign each of the collapsed nodes to a different PE.

Figure 4 shows an example of this approach. The utilization time annotated in the nodes suggest that these two paths can be collapsed as depicted.

To extend the capabilities of this transformation, we may move delays existing in the dependence graph up or downstream. In this manner, we may manipulate the nodes of the graph so that identical paths, suitable for collapsing, have different utilization times. Examples of this situation are shown in section 4.5.

4.4 Removing Input Data Bottleneck

There is another approach towards solving the problems created by a restricted I/O bandwidth. If input pads do not provide enough capability to transfer all needed data at once, the data input process becomes a bottleneck in the implementation. An identical situation is true for output data and output pads. In such cases, it is possible to separate the computation from the transfer of data into or out of the processing structure so that one instance of the algorithm may be under computation, while data belonging to another one is being transferred to/from the system.

Figure 5 depicts this situation. In this figure, three data elements are used three times to

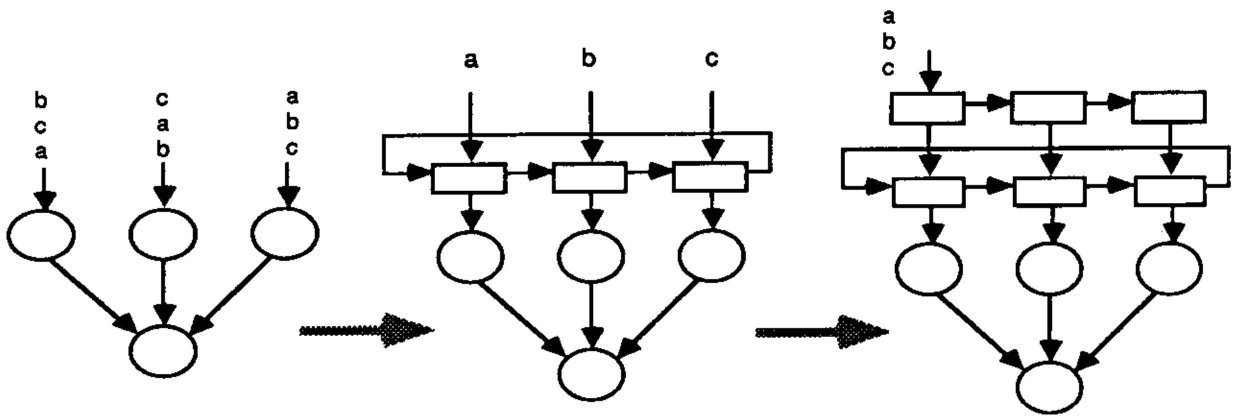

Figure 5: Removing input data bottleneck

perform independent computations in three different units. For the fully-parallel implementation three different input pads are needed, all of them carrying the same data but in a different order. Consequently, the data can be input once, and circulated through a ring structure. If a single input port exists, it can be used to input the data serially through a shift-register structure. After the three elements have been loaded into the shift-register, they are transferred simultaneously to the ring. While these elements are used for computation, a new set of data values is brought into the structure.

The approach requires additional memory elements to store the data, but allows to achieve the degree of parallelism permitted by the dependence graph under a restricted I/O bandwidth situation. Those memory elements can be organized in such a way as to reduce the interconnection requirements of the computation, as shown by the shift-register in Figure 5.

4.5 Application of the Transformations

In this section, we illustrate the application of the proposed methodology to matrix multiplication and LU-decomposition. We show that the application of different transformations to the dependence graph of an algorithm may lead to entirely different architectures. We present the dependence graph of these widely used matrix computations, and use them as target for the application of the methodology outlined here.

4.5.1 Matrix Multiplication Algorithm

The algorithm for multiplication of square matrices is described by the dependence graph in Figure 6. 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.

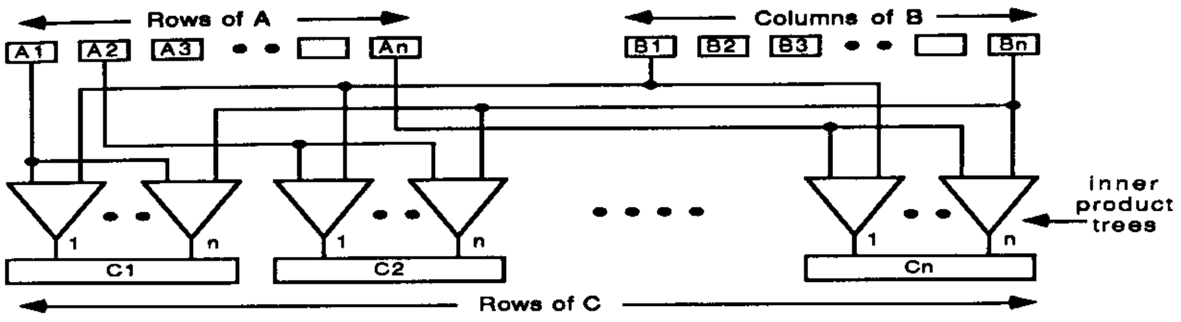

Figure 6: Matrix multiplication dependence graph

- a pipelined implementation of the graph has delay $O(logn)$ and throughput $T = 1[eval/cycle]$. Such implementation requires an I/O bandwidth $O(n^2)$, and broadcasting of the input data.

Several schemes have been proposed for matrix multiplication. Among the most popular implementations, we can list the hexagonally connected mesh proposed by H.T. Kung [19], and the quadratic array suggested by Speiser and Whitehouse [5]. Neither of these schemes has resulted from a systematic evaluation of the various implementation parameters.

4.5.2 I/O and PE Utilization as Main Restrictions in Matrix Multiplication

We assume that the I/O bandwidth is restricted by the implementation to $O(n)$. For simplicity, we consider this restriction as $3n$: one row or column from each of the input matrices, and one row of the result. In such case, it is not possible to exploit the entire parallelism available in the matrix multiplication algorithm due to lack of data. The minimum computation time is now $O(n + (logn))$, since the entire matrix has to be loaded in n cycles. The throughput is now $O(n)$. The implementation of this graph is data-bound.

To take the I/O restriction into account, we modify the dependence graph by restricting output pads first as indicated in section 4.1. We introduce additional nodes to represent the output pads and delays to avoid conflicts, as shown in Figure 7 for a $n = 3$ square matrix. Since all paths in the graph are identical, we use our knowledge about the algorithm and place the delays in main-diagonal-wise fashion (column-wise or row-wise approaches are also possible).

A pipelined implementation of the modified graph has low utilization of units because, in spite of the delays added, all nodes in the graph are evaluated simultaneously but only once every n cycles. To improve the utilization, we move the delays upstream in the dependence graph, as shown in Figure 8. The utilization times annotated in this graph indicate that now nodes are computed at different times. We collapse identical paths with different utilization times and serialize their input data, as proposed in section 4.3. The resulting graph, shown in Figure 9, illustrates the need to replicate data since the same data elements are needed either in different places and/or at different times.

Data duplication may be eliminated in this case in a simple manner, because the interconnection pattern of the replicated data elements is simple. The elements of matrix B are replicated in time so that a single copy of such data is needed, which is accessed repeatedly. The elements of matrix

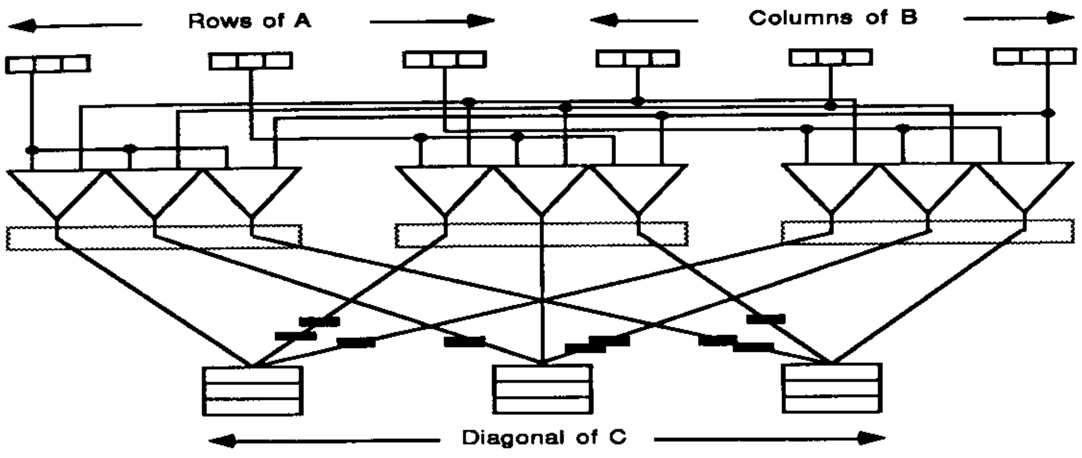

Figure 7: Modified matrix multiplication algorithm for $n = 3$: output pads added

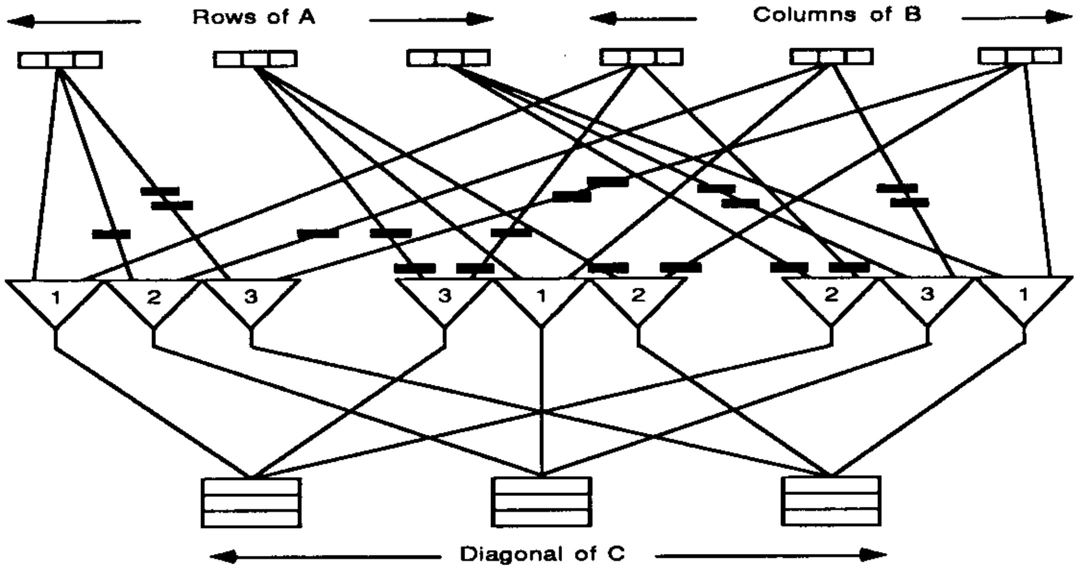

Figure 8: Modified matrix multiplication algorithm: delays moved upstream

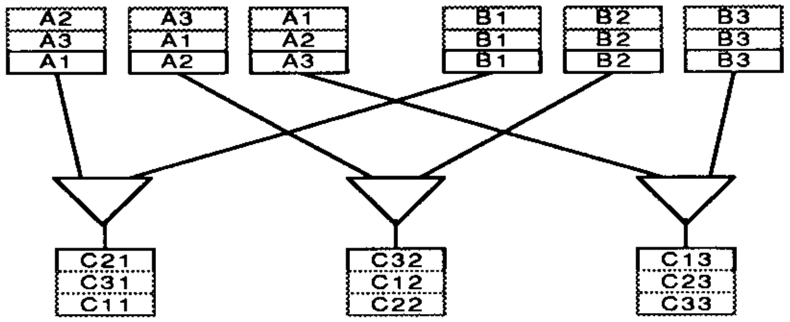

Figure 9: Modified matrix multiplication algorithm: nodes collapsed

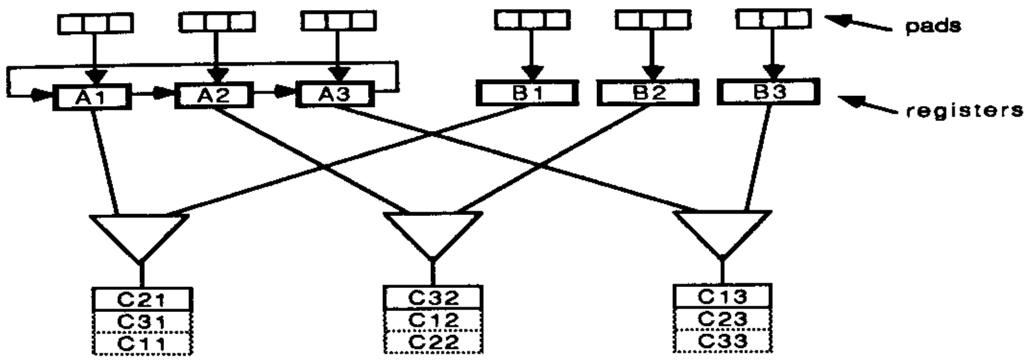

Figure 10: Modified matrix multiplication algorithm: replicated data reduced

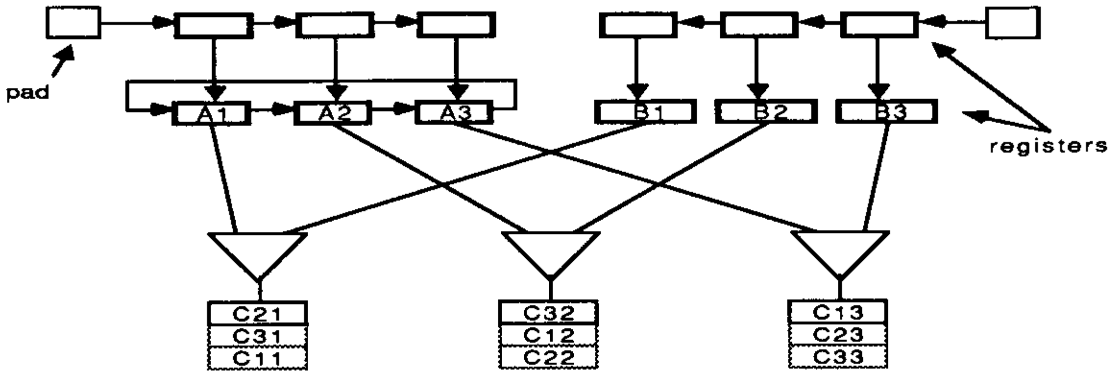

Figure 11: Modified matrix multiplication algorithm: input data bottleneck

A, on the other hand, exhibit a circular–shifting behavior. Storing this data in registers with the same interconnection characteristics allows to keep only one copy of the data. Figure 10 shows the structure resulting after these issues have been considered.

The dependence graph in Figure 10 has input bandwidth $O(n^2)$, but input pads are used only once every n cycles. Reducing the input pads to n produces an input data bottleneck situation, which can be eliminated as suggested in section 4.4; that is, by isolating the input data process from the computation. The modified graph is shown in Figure 11.

A pipelined implementation of the resulting graph has input data bandwidth requirements within the constrains stated, and maximum utilization of the PEs. The operation of the system is as follows: a pair of matrices is loaded in n cycles. After such matrices are stored in the input registers, they are transferred simultaneously to the inner–product trees registers where they are used for the next n cycles. In the meantime, another pair of matrices is being loaded in the input registers. We call this a Multiple Tree architecture for matrix multiplication.

4.5.3 Regularity as Main Restriction in Matrix Multiplication

In this example, we are concerned with regularity in the design, nearest–neighbor communications, no data broadcasting, and identical computation units.

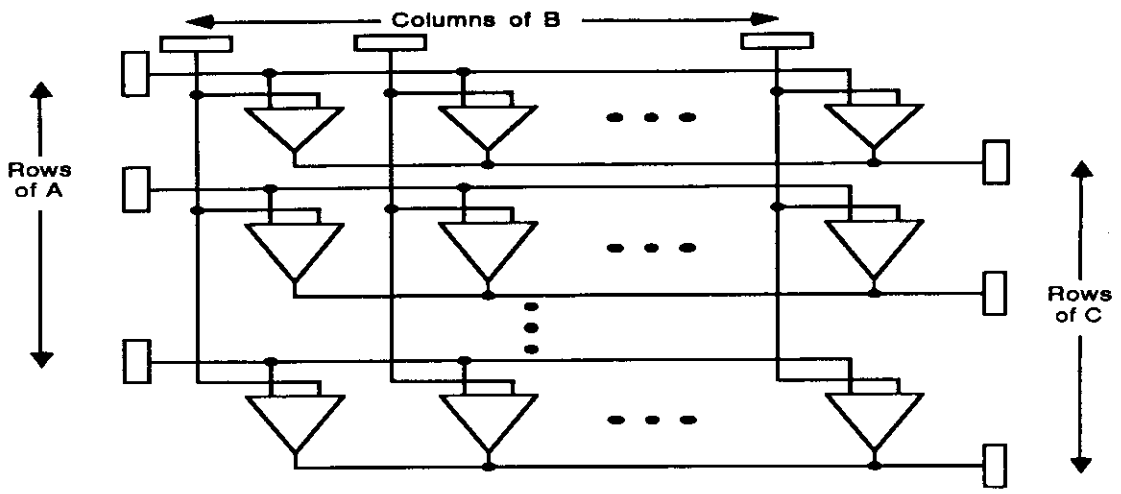

Figure 12: Two-dimensional matrix multiplication graph

Since data dependences in the matrix multiplication algorithm are defined by the two input matrices, we first draw the algorithm as shown in Figure 12. This is exactly the same as Figure 6, but drawn with a two-dimensional input data flow.

In this case, we first attempt to eliminate data broadcasting by replacing it with data pipelining, as suggested in section 4.2. Since broadcasted data paths are of the same length, we take advantage of the knowledge about the algorithm and place the delay nodes in row-wise and column-wise ordering. Figure 13 shows the result of applying the method to the graph in Figure 12. An implementation of this graph has input bandwidth $O(n^2)$, computation time $(2n - 1)$, optimal efficiency, and throughput $O(1)$.

Let's assume now that the input bandwidth of the implementation is restricted to $2n$. Applying the methodology described in section 4.1, we introduce nodes for the input pads and serialize the data input process. As a consequence, we have created an input data bottleneck, and nodes have different utilization time. We choose to collapse identical paths in the inner-product trees, as shown in Figure 14. (Alternatively, we could attempt to isolate the data input process from the computation, as suggested in section 4.4.) Thus, the parallel-input inner-product trees are transformed into serial-input trees. The resulting graph is shown in Figure 15; it has bandwidth $3n$, computation time $(3n-2)$, optimal efficiency, and throughput n. It corresponds to Speiser and Whitehouse's systolic array for matrix multiplication [5].

Thus, we have shown that a systematic transformation of the matrix multiplication dependence graph allows to obtain implementations with different architectures and performance characteristics.

4.5.4 LU-Decomposition Algorithm

The LU-decomposition computation is described by the dependence graph shown in Figure 16. This graph depicts a varying degree of parallelism at different steps in the computation, and broadcasting of intermediate results. Input data, on the other hand, is used in only one specific place.

From the graph, we can infer that this algorithm has a dependence structure quite different

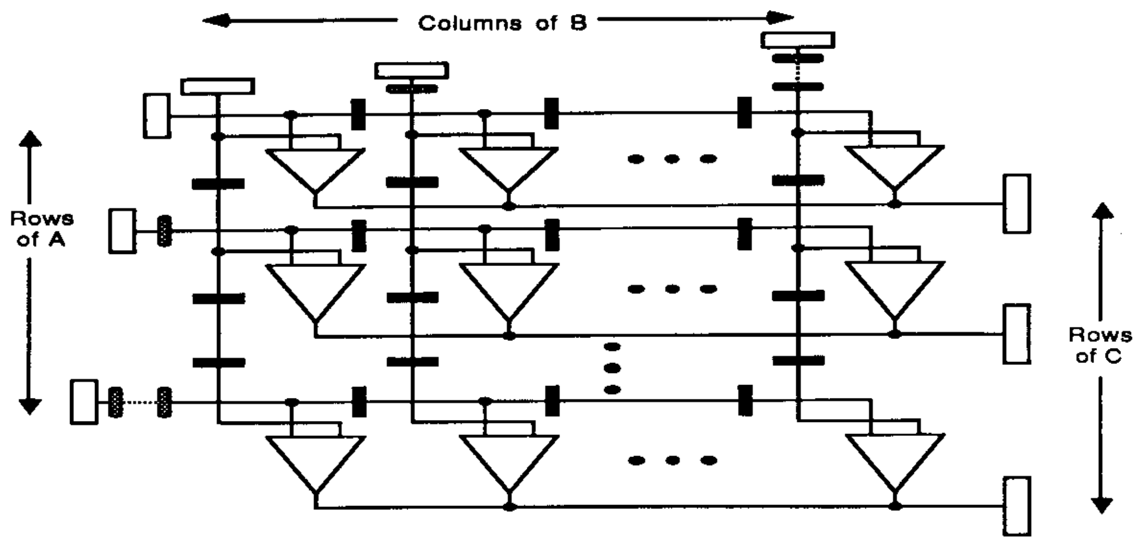

Figure 13: Two-dimensional matrix multiplication graph:delays added

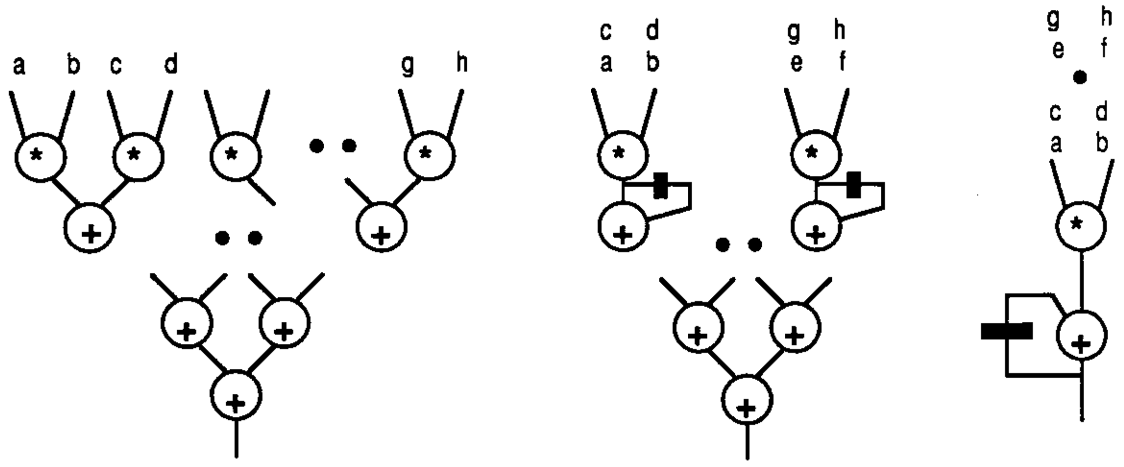

Figure 14: Transformation of inner-product trees

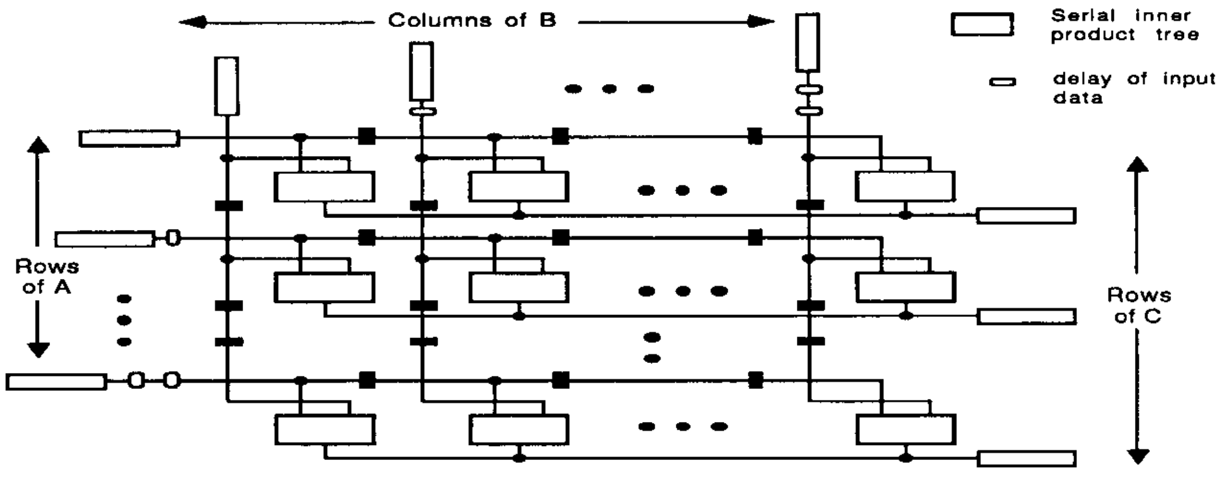

Figure 15: Two-dimensional matrix multiplication graph: serial input trees

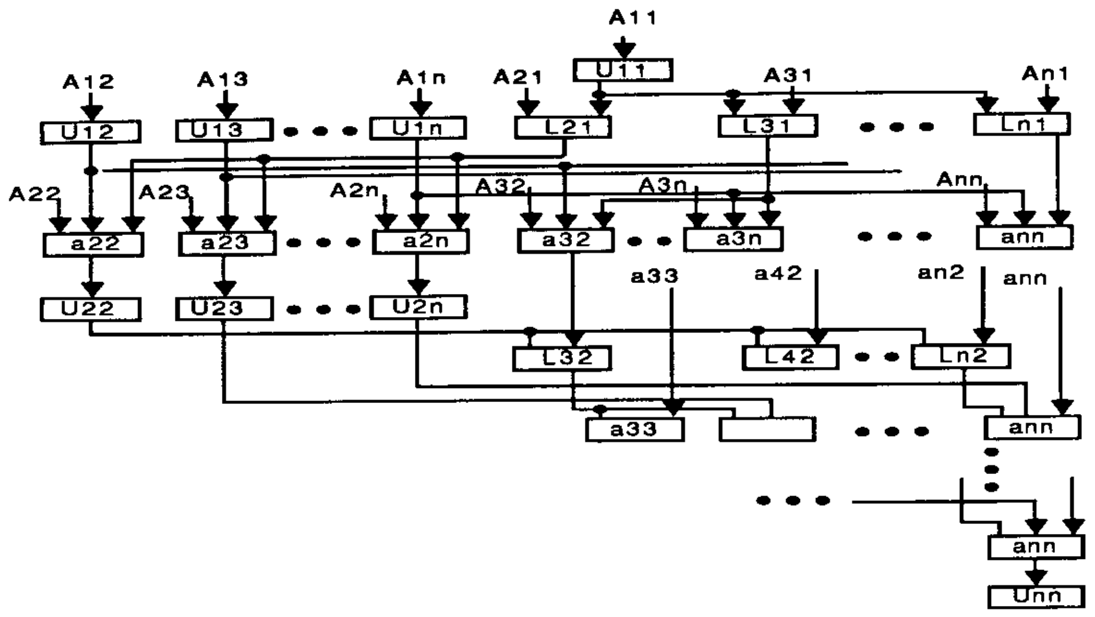

Figure 16: LU–decomposition graph

from matrix multiplication. However, when expressed as recurrence equations, both matrix multiplication and LU–decomposition have similar expressions. This fact has led several researchers to suggest similar arrays for both computations, highlighting their similarity [19], [20]. However, due to the differences, those proposed arrays have low utilization of the PEs.

4.5.5 Data Broadcasting and Utilization as Main Restrictions in LU–Decomposition

In this example, we are interested in the elimination of data broadcasting and in maximizing the utilization of the PEs. We first solve the data broadcasting problem, applying the methodology described in section 4.2. The graph resulting from the application of this transformation to a 5 by 5 matrix is shown in Figure 17, which also indicates the utilization time of the nodes. This graph is characterized by sets of sequential computations (depicted as diagonals of nodes in the figure), 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. No broadcasting is left, but input data is needed throughout the graph.

An implementation of this graph could allocate each set of sequential nodes to a different PE. In such a case, the input matrix needs to be pre–loaded on the array before computation may start. The same scheme was derived through a different methodology in [20]. However, the efficiency of such approach is low, since different processors have widely varying number of operations to perform. For example, the top leftmost PE has only one operation to compute (i.e., only one node of the graph), while the lower rightmost PE has n operations.

From the utilization time for each node in Figure 17, we can see that in each row of the graph only one node is in use at every cycle. Therefore, it is possible to share one PE among the nodes of a row by combining all nodes in each row of the graph into a single one. This corresponds to collapsing paths in the graph where nodes have different utilization times. The resulting graph is shown in

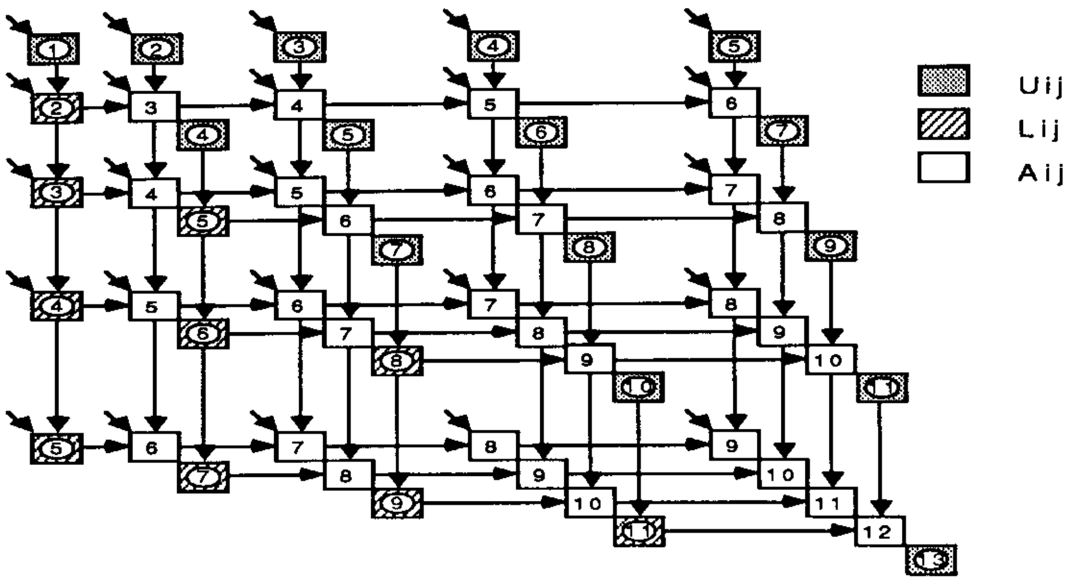

Figure 17: LU–decomposition graph without data broadcasting

Figure 18. A implementation of this graph leads to a triangular array for LU–decomposition, with utilization 0.732, twice that of other square arrays proposed for such computation [19],[20].

5 Conclusions

We have presented a research proposal for the development of a systematic methodology for the design of arrays of PEs for matrix algorithms. The proposed methodology uses a fully–parallel dependence graph of the algorithm as the description tool, because such notation exhibits the optimal features of the algorithm regarding delay, utilization of operation units, and throughput. The methodology consists of the systematic application of transformations on the graph, to fulfill restrictions which render the fully–parallel graph inadequate as an implementation.

We have described the features of a few basic transformations. Their application has allowed us to show that different transformations on a graph may lead to entirely different computing structures for a given algorithm. The selection of an adequate transformation is thus directed by the specific restrictions imposed on the implementation of the algorithm. Those restrictions include issues such as I/O bandwidth, regularity in the interconnection among processing elements (PEs), utilization of those PEs, limited data broadcasting, local communications, data synchronization.

We have presented the application of a preliminary version of this methodology to two known matrix algorithms, namely matrix multiplication and LU–decomposition. We have shown that the approach produces structures which correspond to proposed systolic arrays for these computations, as well as structures which exhibit better efficiency than those arrays. The methodology has been shown as capable to handle and relate features of the algorithm and the implementation, in a unified manner.

Although some of the transformations applied seem of general use, they are not an exhaustive

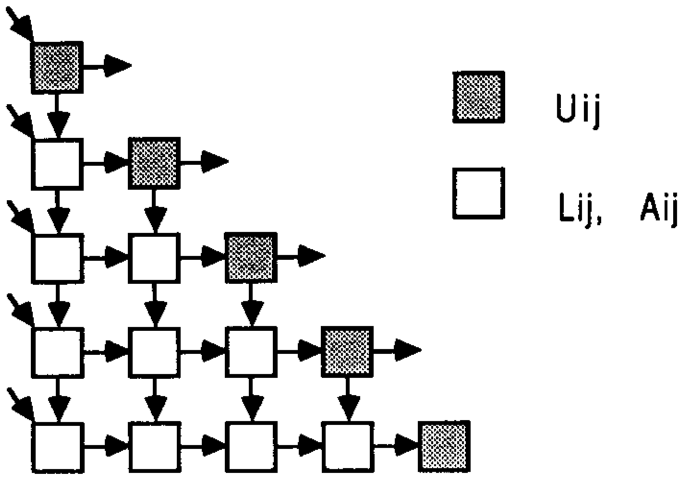

Figure 18: LU–decomposition graph with combined nodes

collection for all matrix computations. In addition, it seems that there are cases where transformations specific to a given algorithm are needed. The objectives of the proposed research include the identification and formal definition of a larger set of transformations, for a more varied class of matrix algorithms than what has been presented here. Additional transformations are also needed to include cases such as the computation of algorithms in arrays smaller than the dimensions of the matrix. A mechanism to define measures of efficiency in the design, and to evaluate implementations according to those measures is also needed. 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 and T. Lang, "Replication and pipelining in multiple instance algorithms," in *International Conference on Parallel Processing*, pp. 285–292, 1986.

[4] J. Moreno and T. Lang, "A multilevel pipelined processor for the Singular Value Decomposition," in *SPIE Real–Time Signal Processing IX*, 1986.

[5] 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.

[6] J. Ullman, *Computational Aspects of VLSI*, ch. 2. Computer Science Press, 1984.

[7] R. Brent and H. Kung, "Area–time complexity of binary multiplier," *Journal of ACM*, vol. 28, no. 3, pp. 521–534, 1981.

[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] J. Blackmer, G. Frank, and P. Kuekes, "A 200 million operations per second (MOPS) systolic processor," in *SPIE Real-Time Signal Processing IV*, pp. 10–18, 1981.

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

[11] G. Li and B. Wah, "The design of optimal systolic arrays," *IEEE Trans. Computers*, vol. C–34, pp. 66–77, Oct. 1984.

[12] I. Ramakrishnan and P. Varman, "An optimal family of matrix multiplication algorithms on linear arrays," in *International Conference on Parallel Processing*, pp. 376–383, 1985.

[13] M. Sheeran, "μFP – an algebraic VLSI design language," Technical Monograph PRG–39, Oxford University Computing Laboratory, University of Oxford, Sep. 1984.

[14] C. Guerra and R. Melhem, "Synthesizing non–uniform systolic designs," in *International Conference on Parallel Processing*, pp. 765–771, 1986.

[15] H. Kung and W. Lin, "An algebra for systolic computation," in *Conference on Elliptic Problem Solvers*, pp. 141–160, 1983.

[16] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *11th Annual Symposium on Computer Architecture*, pp. 208–214, 1984.

[17] M. Chen, "Synthesizing VLSI architectures: dynamic programming solver," in *International Conference on Parallel Processing*, pp. 776–784, 1986.

[18] V. Weiser and A. Davis, "A wavefron notation tool for VLSI array design," in *VLSI Systems and Computations*, (H. Kung et al., ed.), pp. 226–234, Computer Science Press, Oct. 1981.

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

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

[21] J. Moreno and T. Lang, "Design of special–purpose arrays for matrix computations," in *submitted to SPIE Real-Time Signal Processing X*, 1987.

[22] W. Miranker and A. Winkler, "Space–time representations of computational structures," *Computing*, vol. 32, pp. 93–114, 1984.

[23] I. Ramakrishnan, D. Fussell, and A. Silberschatz, "On mapping homogeneous graphs on a linear array–processor model," in *International Conference on Parallel Processing*, pp. 440–447, 1983.

[24] T. Barnwell and D. Schwartz, "Optimal implementation of flow graphs on synchronous multiprocessors," in *Asilomar Conference on Circuits and Systems*, pp. 188–193, 1983.

[25] J. Jover and T. Kailath, "Design framework for systolic–type arrays," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 8.5.1–8.5.4, 1984.

[26] M. Lam and J. Mostow, "A transformational model of VLSI systolic design," *IEEE Computer*, pp. 42–52, Feb. 1985.

[27] R. Chapman, T. Durrani, and T. Willey, "Design strategies for implementing systolic and wavefront arrays using OCCAM," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 292–295, 1985.

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

[29] J. Navarro, J. LLaberia, and M. Valero, "Solving matrix problems with no size restriction on a systolic array processor," in *International Conference on Parallel Processing*, pp. 676–683, 1986.

[30] D. Moldovan, C. Wu, and J. Fortes, "Mapping an arbitrarily large QR algorithm into a fixed size VLSI array," in *International Conference on Parallel Processing*, pp. 365–373, 1984.

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