One of the most challenging tasks in parallel processing is to compile from a high-level functional definition to a complete description of an efficient placement and interconnection of an ensemble of processing elements. The processing elements may have different functionality and their interconnection may be changing dynamically dependent upon the computation itself. In this paper, a design methodology for synthesizing such placements and interconnections is presented. We show how a systolic architecture can be generated systematically by a series of transformations, each aiming at optimizing the target implementation for a specific purpose, from a mathematical problem definition. A set of formal rules of algebraic transformations and their motivations are given. Throughout this paper, the problem of LU decomposition is illustrated and a new design is derived.

Placement and Interconnection of Systolic Processing Elements: A New LU-Decomposition Algorithm

Marina C. Chen

Research Report YALEU/DCS/RR-498
October 1986

Work supported in part by the Office of Naval Research under Contract No. N00014-86-K-0296.
Approved for public release: distribution is unlimited.

(Selected as the best paper in the Architectures area of ICCD '86)

PLACEMENT AND INTERCONNECTION OF SYSTOLIC PROCESSING ELEMENTS: A NEW LU DECOMPOSITION ALGORITHM

Marina C. Chen

Department of Computer Science
Yale University
New Haven, CT 06520
chen-marina@yale

Abstract One of the most challenging tasks in parallel processing is to compile from a high-level functional definition to a complete description of an efficient placement and interconnection of an ensemble of processing elements. The processing elements may have different functionality and their interconnection may be changing dynamically dependent upon the computation itself. In this paper, a design methodology for synthesizing such placements and interconnections is presented. We show how a systolic architecture can be generated systematically by a series of transformations, each aiming at optimizing the target implementation for a specific purpose, from a mathematical problem definition. A set of formal rules of algebraic transformations and their motivations are given. Throughout this paper, the problem of LU decomposition is illustrated and a new design is derived.

1. Introduction

One of the major issues in parallel processing is how to organize a given computation so that, collectively, hundreds of thousands of autonomous parallel processes may accomplish the task in an efficient manner. The technology, on the other hand, constrains the way in which parallel computation can be organized. Dominant costs at the technological level often have profound implications in the designs at algorithmic and architectural levels. Clearly, to achieve an efficient design, one must take advantage of what the technology can offer, and minimize the costs associated with the constraints it imposes upon a design.

The so-called systolic algorithms [10] make use of hardware technology very effectively in several aspects: (1) they avoid the incurrence of high communication costs by organizing the computation in such a way that only local communications take place; (2) they avoid large numbers of fan-ins and fan-outs by using networks of fixed degrees of connectivity; and (3) they use hardware resources effectively by pipelining, i.e., re-using each component $O(k)$ number of times where k is proportional to problem sizes (e.g. linear or square root of problem size). Numerous reports [14, 15], [13], [16], [12], [2], [3], [1], and [7] have appeared in the area of synthesis methodology for systolic algorithms. A detailed review on the relative strength of these different approaches and comparison of the computational complexity of the mapping procedures appears in [3]. Mostly, the emphasis of these reported synthesis methods is on the mapping of a problem specification to a systolic architecture by a linear transform. The specification is mostly given in recurrences of one form or another; very little attention has been paid to how to come up with a suitable problem specification that is amenable to a linear transformation in the first place. In [14] and [13], a problem specification is obtained from a FORTRAN loop by augmenting it with suitable indices. In [3], several guidelines (e.g. first order recursion equations for nearest neighbor communications) but no specific formal rules are given for transforming specifications to a target specification. In [7], the method first seeks linear transforms and then requires the user to modify the problem specification if the former fails. The process iterates until some linear transform is found, but no guideline as to how modification should proceed is given. Clearly, a unified methodology for synthesizing systolic algorithms, at the specification level as well as the mapping level, along with its automation, is highly desirable.

In this paper, we show how a systolic algorithm can be generated systematically by a series of transformations, each aiming at optimizing a target implementation, from a (mathematical) problem definition. A set of formal rules of algebraic transformations and their motivations are given. Throughout this paper, the problem of LU decomposition is used for illustrating synthesis by algorithm transformation. All specifications, from the initial problem definition to the target specification, and the rules of transformations, are described in the same notation — the language **Crystal** [4]. Several key ideas underlie the synthesis approach:

1. Mathematical definition is used as an initial specification, for two reasons: First, the presentation of a definition is aimed at human understanding of the problem and communicating it to others, not programming. Therefore a definition is most suitable as a human interface to computer aided programming tools. Secondly, a problem definition is a "pure" specification without being contaminated by constraints at the implementation level: it is naturally parallel. Sequentiality is an example of a constraint in conventional programming on a single processor machine. Without the presence of such artificial constraints, a mathematical definition can be interpreted as an algorithm which exhibits extreme parallelism.
2. However, the parallel implementation media impose their own set of constraints on the design of algorithms, and the extreme parallelism exhibited in the definition may not be most efficient from the standpoint of parallel implementation. For instance, bounded fan-in and fan-out degrees and locality of communications are often imposed by realistic parallel implementations such as VLSI architectures or programs on a network of parallel processors. Extreme parallelism without taking into account these constraints may turn out not to be efficient. The heart of the synthesis approach described here lies in harnessing the extreme parallelism exhibited in the definition by infusing suitable data dependencies into the problem definition so as to optimize the efficency of the resulting parallel implementation. It is an opposite approach to the one that extracts parallelism from sequential

programs.

3. Source program and target program optimizations are employed in the process of devising efficient parallel solutions to problems. A mathematical definition which is conceptually clear and easy to express by the user does not guarantee its efficiency in implementation. Such optimizations relieve a user from many implementation details. On the other hand, certain optimization is concerned with the trade-offs at the implementation level, and hence must be performed at the target program level. In both cases, the transformation rules for optimization and the programs themselves are specified in the same high-level language **Crystal**.
4. Pipelining is a form of efficient resource sharing in parallel processing. It can be automatically incorporated into algorithms by a process called *space-time mapping*.
5. The design space for parallel algorithms has been enlarged from the one-dimensional space of sequential programming to multi-dimensional space in which processors can be arranged in a variety of different ways. The number of possible solutions to a given problem can be different not only in the algorithmic sense: the same algorithm can have many different realizations in multi-dimensional space and time, each with different control and data flow and space-time tradeoffs. These different space-time realizations can be systematically obtained.
6. The design methodology advocates a uniform framework in which parallel algorithms and architectures are described, from source level to target level, in the same language **Crystal**. Such general purpose parallel programming environment allows the compilation methods and optimization techniques described in this paper, or any other new methods and techniques be integrated readily within the existing framework, and makes them independent of the target machine architectures or implementation media. Furthermore, a simulation of a target VLSI architecture or a parallel program is automatically generated by executing the **Crystal** program in any given implementation of **Crystal**.

The rest of the paper is organized as follows: In Section 2, the mathematical definition of LU decomposition is given. In Section 3, a large number of fan-ins and fan-outs, and long range communications in a parallel algorithm are identified as features that incur high costs in hardware, and hence should be avoided. Transformations of algorithms to reduce the fan-ins and fan-outs to bounded degrees, and to reduce long range communications to local ones, are introduced. In Section 4, program optimization at the source level is illustrated. In Section 5, space-time mapping which incorporates pipelining into an algorithm is described and illustrated. In Section 6.3, optimization of control signals at the target program level is illustrated.

2. Definition of LU Decomposition

2.1. Recursion equations

LU decomposition of matrices can be defined as the following system of recursion equations which is essentially the standard definition of LU decomposition except for slight notational variances. It describes that the decomposition of the matrix A is obtained iteratively as index k ranges from 0 to n, and at each iteration a column of the L matrix and a row of the U matrix are computed.

In the system of equations, we call i, j, and k indices; they correspond to subscripts (e.g. indices of matrix elements) and superscripts (e.g. indices for iterations) in the linear recurrence notation*. They always assume discrete values, and these values are referred to as indices. We refer to a, L, and U as identifiers of functions.

$$a(i,j,k) = \begin{cases} k = 0 \rightarrow A(i,j) \\ 0 < k \le n \rightarrow a(i,j,k-1) + L(i,k) \times (-U(k,j)) \end{cases}$$

$$L(i,k) = \begin{cases} 1 \le i < k \rightarrow 0 \\ i = k \rightarrow 1 \\ k < i \rightarrow a(i,k,k-1) \times U(k,k)^{-1} \end{cases}$$

$$U(k,j) = \begin{cases} 1 \le j < k \rightarrow 0 \\ k \le j \rightarrow a(k,j,k-1) \end{cases} \tag{2.1}$$

2.2. Parallel interpretation of recursion equations

The definition of LU decomposition above can be interpreted as a parallel program as follows: Each triple of indices (i,j,k) is interpreted as a *process* in a domain of process $P \stackrel{\text{def}}{=} D \times D \times D$, the Cartesian product of the domain of the indices $D \stackrel{\text{def}}{=} \{i : 0 \le i \le n, i \text{ is an integer}\}$.

The domain of process P is ordered by a data dependency relation. We say that a process **u** immediately precedes **v** ($\mathbf{u} \prec \mathbf{v}$), or that **v** immediately depends on **u** ($\mathbf{v} \succ \mathbf{u}$), when **v** appears on the left-hand side of a system and **u** appears on the right-hand side of a system. A system (e.g. 2.1) must be such that the transitive closure "$\overset{*}{\prec}$" (precedes) of "$\prec$" on all processes is a partial order, and there is no infinite decreasing chain from any node **v**, nor is there an infinite number of processes that immediately precede **v**. The set $(P, \overset{*}{\prec})$ can be depicted as a DAG (Directed Acyclic Graph), which is shown in Figure 1. It consists of nodes, where each node corresponds to a process (i,j,k) in the set P, and directed edges, where an edge is emitting from node **u** to node **v** if $\mathbf{u} \prec \mathbf{v}$. Those nodes that have no incoming edges are called *sources*.

The processes are parallel in nature, a computation starts at the sources which are properly initialized, and is followed by other processes each of which starts execution when all of its required inputs, or dependent data, become available.

2.3. Local processing functions

The local function each process should perform can be extracted from the definition. The local function performed may vary from process to process; hence, often a family of local functions, subscripted by the indices of the process, are defined. For LU decomposition, three families of processing functions $F_{i,j,k}$, $G_{i,k}$, and $H_{j,k}$ listed below are defined. For instance, the family of functions $F_{i,j,k}$ is defined with three formal parameters x, y, z and a constant function A, and it is defined as a composition of primitive functions such as addition, multiplication, boolean predicates, and conditionals.

$$F_{i,j,k}(x,y,z) \stackrel{\text{def}}{=} \begin{cases} k = 0 \rightarrow A(i,j) \\ 0 < k \le n \rightarrow x + y \times (-z), \end{cases}$$

$$G_{i,k}(x,y) \stackrel{\text{def}}{=} \begin{cases} 1 \le i < k \rightarrow 0 \\ i = k \rightarrow 1 \qquad \text{for all } 0 < j \le n \\ k < i \rightarrow x \times y^{-1}, \end{cases} \tag{2.2}$$

$$H_{j,k}(x) \stackrel{\text{def}}{=} \begin{cases} 1 \le j < k \rightarrow 0 \\ k \le j \rightarrow x, \end{cases} \qquad \text{for all } 0 < i \le n$$

*The choice of a functional notation such as **Crystal**, in which subscripts and superscripts are treated uniformly, is not only for linguistic reasons, but also because, conceptually, they deserve a symmetrical treatment. In a recurrence, a subscript such as i or j is thought to be an index for a static location of some structured data, and a superscript such as k is thought to be an index for iterative computational steps. However, in the multi-dimensional design space of parallel programs, there are numerous different ways a subscript or superscript can be interpreted: statically as a location, dynamically as a computational step, or as a linear combination of both.

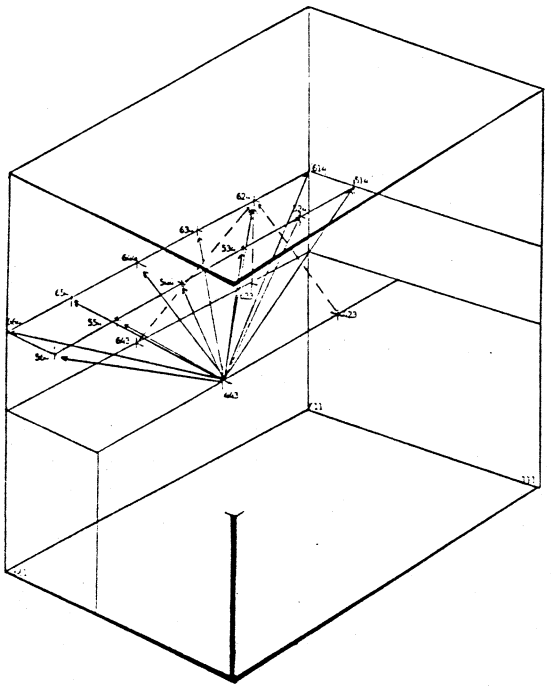

Figure 1: The DAG describing the data dependency of process $(6.2.4)$ and the fan-outs of process $(4,4,3)$.

2.4. Communication functions

Similarly, the communications between processes can be extracted from the definition. Each formal parameter of a local processing function, at run time, might either be substituted by a functional value evaluated within the process, or by a value passed on by other processes. In the latter case, a communication function is defined for specifying which process is passing on a value for the parameter substitution. Listed below are the families of communication functions extracted from the definition of LU decomposition.

$$\begin{aligned} \tau_{1,i,j,k} &\stackrel{\text{def}}{=} (i,j,k-1), \\ \tau_{2,i,k} &\stackrel{\text{def}}{=} (i,k,k-1), \text{ for all } 0 < j \leq n \\ \tau_{3,j,k} &\stackrel{\text{def}}{=} (k,j,k-1), \text{ for all } 0 < i \leq n \end{aligned} \tag{2.3}$$

For example, from the definition of $H_{j,k}$ and System (2.1), the formal parameter x of local processing function $H_{j,k}$ in process (i,j,k) should be replaced by $a(k,j,k-1)$, which is passed by process $(k,j,k-1)$; hence the definition of $\tau_{3,j,k}$.

Note that all of the above communication functions do not take any arguments other than the constants i, j, and k which are associated with a process. In other words, the communication specifies a data flow that is independent of input or data obtained during computation. In the algorithm for LU decomposition with pivoting, one of the communication functions becomes data dependent.

2.5. Complexity of the naive parallel program

Using one processor for each process in System (2.1), we obtain immediately a naive parallel program in which the local processing at each processor and the communications between processors are obtained from the definition as described above. Altogether $O(n^3)$ number of parallel processors are needed. As shown in System (2.1), any value $a(i,j,k)$ cannot be computed until $a(i,j,k-1)$ is computed, and hence the number of time steps needed to compute the result is n.

3. Implementation Constraints Affecting Efficiency

3.1. Large Fan-in and fan-out degrees

Due to the inherent physical constraints imposed by the driving capability of communication channels, power consumptions, heat dissipations, memory bandwidth, etc., data cannot be sent or received to or from a large number of destinations or sources in a unit communication time. In other words, any program should avoid large fan-in and fan-out degrees, where the *fan-in degree* of a datum is defined as the number of data items it depends on, and *fan-out degree* is the number of data items dependent upon it.

3.2. Counting fan-in and fan-out degrees

The fan-in degree of a process (i,j,k) is obtained by counting the number of distinct processes (i',j',k') appearing on the right hand side of a definition. Conversely, the fan-out degree of a process (i,j,k) is the number of times (i,j,k) appears on the right-hand side while a process (i',j',k'), each time a different (i',j',k'), appears on the left-hand side.

For LU decomposition, by definition (2.1), the fan-in degree of processes $(i,j,0)$ is zero, because it is a source. The fan-in degree of processes (i,j,k) where $k > 0$, $i > k$, and $j \geq k$ is 4, since it depends on values computed at processes $(i,j,k-1)$, $(i,k,k-1)$ (via definition of $L(i,k)$), $(k,j,k-1)$ (via definition of $U(k,j)$), $(k,k,k-1)$ (via definition of $U(k,k)$), as shown in Figure 1. The fan-out degree of processes $(k,k,k-1)$ for all $k > 0$ deserves special attention. Each value $a(k,k,k-1)$ is needed by processes (i,j,k) for all $k < i \leq n$ and $0 < j \leq n$, and thus its fan-out degree is $(n-i) \times n$, which is proportional to the square of the problem size n.

3.3. Reduction to bounded fan-in and fan-out degrees

Since all processes appearing in System (2.1) have constant fan-in degrees, no transformation aiming at fan-in degree reduction is needed.

3.3.1. Concurrent assignments vs. serial assignments

On the other hand, reduction of the fan-out degree must be performed. It can be achieved by serializing an n-destination communication as a series of nearest neighbor communications. The following two rules of transformations are used:

Proposition 3.1. *Let $z(l)$ for $u \leq l \leq v$ where u and v are integers and $u < v$ be $v-u+1$ variables to which a value x shall be assigned, i.e., $z(l) = x$ for $u \leq l \leq v$ (where the fan-out degree of x is $v-u+1$), then the assignments of these variables can be performed instead by recursion in index j:*

$$z(l) = \begin{cases} l = u \rightarrow x \\ l > u \rightarrow z(l-1) \end{cases} \tag{3.1}$$

or by

$$z(l) = \begin{cases} l = v \rightarrow x \\ l < v \rightarrow z(l+1) \end{cases} \tag{3.2}$$

for $u \leq l \leq v$

where the fan-out degrees of x and $z(l)$ for $u \leq l \leq v$ are 1.

The two recursion equations above are only two out of many possible ways a value x can be concurrently assigned to many variables $z(l)$. The first equation describes the situation where x is assigned first to the variable at the endpoint u of the interval between u and v. Alternatively, the second equation describes the assignment starting at the other endpoint v. Similarly, x could be first assigned to a $z(w)$, where w is somewhere in the middle of the interval. To complete the concurrent assignments, all of the above schemes, called linear shift arrays, require time steps linear to the number of variables to

be assigned. A much different way of implementing the concurrent assignment of x to many variables may be via a broadcasting tree, in which only a logarithmic number of steps are needed to complete the assignments. Interested readers may try to describe the broadcasting tree by recursion equations. Such transformation is very useful in devising efficient parallel programs. The question now is which rule should be selected for transformation.

3.3.2. Time complexity and program indices

For the LU decomposition problem, in order to decrease the fan-out degree of $L(i,k)$, the schemes of linear shift array and the broadcast tree are first compared. Recall that System (2.1) takes n steps to complete, where index k counts the number of steps. If the time steps of the new algorithm can be counted by any expression of the indices only to the first power, then the time complexity of the algorithm is maintained at $O(n)$ due to the range of the three indices. For this reason, the linear shift array is chosen because it increases the time complexity no more than a constant factor and requires less hardware than a broadcasting tree.

By matching the indices of $a(i,j,k)$ on the left-hand side of System (2.1) with those of $L(i,k)$ on the right, it is clear we should define a new variable $l(i,j,k)$ to which the value $L(i,k)$ is to be assigned for each process (i,j,k). Proposition 3.1 should be applied to variables $l(i,j,k)$, with recursion in j. Similarly, we define for each process (i,j,k), a variable $u(i,j,k)$ to which the value $U(k,j)$ is to be assigned, and apply the proposition to $u(i,j,k)$ with recursion in index i.

Within the class of linear shift arrays there are a few possible choices: initial assignments to variables at endpoints of an interval are in general preferred since they result in uni-directional data flow, which requires simpler control than bi-directional data flow. However, locality is another factor:

3.3.3. Locality of initial assignment

In choosing the lower and upper bounds u and v for index l, and the particular variable $z(l)$ to which the value x should be initially assigned in the Proposition 3.1, one must be aware of the issue of locality of communications. For example, in the case when $i > k$, $L(i,k)$ is defined in terms of $a(i,k,k-1)$ and $a(k,k,k-1)$ (via $U(k,k)$), while in the case of $i \le k$, it is defined by two constants. To assure that communications are local (in fact, between nearest neighbor), for the case $i > k$, $l(i,k,k)$ should be chosen as the variable to which $L(i,k)$ is initially assigned. On the other hand, as a convenience at the level of implementation, constants should be assigned as an intial condition in the equations. Hence for $i \le k$, $L(i,k)$ should be initially assigned to $l(i,0,k)$.

3.3.4. Algebraic transformations

Putting all of the above considerations together, the transformations proceed as follows:

1. The definition of $L(i,k)$ is split into two cases:

$$L_1(i,k) = \begin{cases} i < k \to 0 \\ i = k \to 1 \end{cases}$$
$$L_2(i,k) = a(i,k,k-1) \times U(k,k)^{-1}$$

2. For the case $k < i$, first Rule (3.1) in Proposition 3.1 is applied to $l(i,j,k)$ and $L_2(i,k)$ for j in the range $k \le j \le n$, and then Rule (3.2) is applied for $1 \le j \le k$, which result, respectively, in:

$$l(i,j,k) = \begin{cases} j = k \to L_2(i,k) \\ k < j \le n \to l(i,j-1,k) \end{cases}, \text{ for } k \le j \le n, \text{ and} \quad (3.3)$$
$$l(i,j,k) = \begin{cases} j = k \to L_2(i,k) \\ 1 \le j < k \to l(i,j+1,k) \end{cases}, \text{ for } 1 \le j < k. \quad (3.4)$$

for $i > k$.

In the case $i \le k$, Rule (3.1) in Proposition 3.1 is applied to $l(i,j,k)$ and $L_1(i,k)$ for j in the range $0 \le j \le n$:

$$l(i,j,k) = \begin{cases} j = 0 \to L_1(i,k) \\ 0 < j \le n \to l(i,j-1,k) \end{cases}, \text{ for } 0 \le j \le n$$

The fan-out degrees of $L(i,k)$ ($L_1(i,k)$ and $L_2(i,k)$) are now reduced to 2 and that of each $l(i,j,k)$ is reduced to 1.

Similarly, let $u(i,j,k)$ be the variable at each process (i,j,k) to which $U(j,k)$ is to be assigned and let

$$U_1(k,j) = 0$$
$$U_2(k,j) = a(k,j,k-1).$$

In the case $k \le j \le n$, Rule ((3.1)) is applied for $k \le i \le n$ and Rule ((3.2)) for $1 \le i \le k$. In the case $1 \le j < k$, Rule ((3.1)) is applied for $0 \le i \le n$.

3. The three equations defining $l(i,j,k)$ and, similarly, those defining $u(i,j,k)$ are merged and then substituted into the original system of equations (2.1) for $L(i,k)$ and $U(k,j)$ on its right-hand side. Moreover, the definitions of $L_1(i,k)$, $L_2(i,k)$ $U_1(k,j)$, and $U_2(k,j)$ are substituted directly into the equations, and we obtain a new definition of LU decomposition that has bounded fan-in and fan-out degrees:

$$a(i,j,k) = \begin{cases} k = 0 \to A(i,j) \\ 0 < k \le n \to a(i,j,k-1) + l(i,j,k) \times (-u(i,j,k)) \end{cases}$$
$$l(i,j,k) = \begin{cases} i \le k \to \begin{cases} j = 0 \to \begin{cases} i < k \to 0 \\ i = k \to 1 \end{cases} \\ 0 < j \le n \to l(i,j-1,k) \end{cases} \\ k < i \le n \to \begin{cases} j = k \to a(i,j,k-1) \times u(i,j,k)^{-1} \\ k < j \le n \to l(i,j-1,k) \\ 1 \le j < k \to l(i,j+1,k) \end{cases} \end{cases}$$
$$u(i,j,k) = \begin{cases} j < k \to \begin{cases} i = 0 \to 0 \\ 0 < i \le n \to u(i-1,j,k) \end{cases} \\ k \le j \to \begin{cases} i = k \to a(i,j,k-1) \\ k < i \le n \to l(i-1,j,k) \\ 1 \le i < k \to l(i+1,j,k) \end{cases} \end{cases} \quad (3.5)$$

A remaining criterion for a good parallel algorithm is that all communications between processes defined by a system of recursion equations should be local. The more local a communication is, the

less the area in hardware and the shorter the time required for its implementation. Since System (3.5) contains only local communicationsl, no transformation is needed. Techiques for reducing the range of communications in general, please see [5].

4. Program Optimization

At the level of problem definition, what is conceptually clear and easy to express by a user does not guarantee the efficiency in its implementation. Program optimizations at the source level are needed, for example, to simplify program control, reduce the complexity of operations in each processor, maintain a balance between communications and computations, etc.

4.1. Simplify program control

The system of equations often has many alternative branches on the right hand side of each equation specifying different processing functions and communication functions. The execution of such a program requires control information which determines one of the alternatives to be actually carried out. The fewer number of different processing functions and communication functions, the less control is needed. Reducing control information to a minimum level is of particular importance for a VLSI implementation of such a program due to the area that must be devoted to such control.

The program in Equation (3.5) can be improved by the elimination of some of the branches on its right-hand side. Observe that whenever $i < k$, $l(i,j,k)$=0 and whenever $j < k$, $u(i,j,k) = 0$. If the binary multiplication operation is defined so that whenever a zero appears in one of the arguments, the result is zero irrespective of the other arguments, then we don't care what the value of $u(i,j,k)$ is in the former case and similarly, what the value of $l(i,j,k)$ is in the latter case. Since they are "don't care" values, their definition in these cases can be redefined so that some of the right-hand side branches can be merged with others and the total number of branches in the system becomes smaller. The merging of the branches for the definitions of $l(i,j,k)$ and $u(i,j,k)$ are illustrated in **Table 1.**

4.2. Detect common expensive operations

Similar to the idea of detecting common sub-expressions in conventional compilation, detecting *common expensive operations* can reduce the total cost of a parallel implementation. An expensive operation is, for instance, one that might take more time to complete than other operations in an otherwise balanced system. It then becomes necessary to devote powerful computing resources to this operation so as to keep up the performance. If a given expensive operation occurs commonly in quite a few processes in the system, then the alternative of communicating the result of the operation in one process to other processes may reduce the total cost of the computation.

Such an optimization technique can be applied at the source level as illustrated by the following example. System (3.5) contains three data streams: stream a for the matrix to be decomposed, and streams l and u containing, respectively, the lower and upper triangular matrices to be obtained. Its domain of processes D^3 is three-dimensional. Table 2 contains the part of the program after optimization, which is a better program due to the reduction in the number of division operations, from order n^2 to n. The modification to the program to achieve the optimization consists of moving the processes performing divisions in a region of the half plane $i > k = j$ to those in a segment of the line $i = j = k$ by using an extra data stream b which flows along the same direction as stream u. Stream b can be sent on the same channel as u.

Incorporating the above two transformations, using $\perp$ for an undefined value, Equation 3.5 becomes:

$$
\begin{aligned}
a(i,j,k) &= \begin{cases} k = 0 \rightarrow A(i,j) \\ 0 < k \le n \rightarrow a(i,j,k-1) + l(i,j,k) \times (-u(i,j,k)) \end{cases} \\
l(i,j,k) &= \begin{cases} i \le k \rightarrow \begin{cases} j = 0 \rightarrow \begin{cases} i < k \rightarrow 0 \\ i = k \rightarrow 1 \end{cases} \\ 0 < j \le n \rightarrow l(i,j-1,k) \end{cases} \\ k < i \le n \rightarrow \begin{cases} j = k \rightarrow a(i,j,k-1) \times b(i,j,k) \\ k \ne j \le n \rightarrow l(i,j-1,k) \end{cases} \end{cases} \\
[u,b](i,j,k) &= \begin{cases} j < k \rightarrow \begin{cases} i = 0 \rightarrow [0, \perp] \\ 0 < i \le n \rightarrow [u(i-1,j,k), \perp] \end{cases} \\ k = j \rightarrow \begin{cases} i = k \rightarrow [a(i,j,k-1), a(i,j,k-1)^{-1}] \\ i \ne k \rightarrow [u(i-1,j,k), b(i-1,j,k)] \end{cases} \\ k < j \rightarrow \begin{cases} i = k \rightarrow [a(i,j,k-1), \perp] \\ i \ne k \rightarrow [u(i-1,j,k), \perp] \end{cases} \end{cases}
\end{aligned}
\tag{4.1}
$$

left-hand side	right-hand side	
	before	after
$l(i,j,k)$	$k < i \rightarrow \begin{cases} k < j \rightarrow l(i,j-1,k) \\ j < k \rightarrow l(i,j+1,k) \end{cases}$	$k < i \rightarrow j \ne k \rightarrow l(i,j-1,k)$
$u(i,j,k)$	$k < j \rightarrow \begin{cases} k < i \rightarrow u(i-1,j,k) \\ i < k \rightarrow l(i+1,j,k) \end{cases}$	$k < j \rightarrow i \ne k \rightarrow u(i-1,j,k)$

Table 1: Program optimization by branch merging.

left hand side	conditions	right hand side	
		before	after
$l(i,j,k)$	$(j = k) \wedge (i > k)$	$a(i,j,k-1) \times [u(i,j,k)]^{-1}$	$a(i,j,k-1) \times b(i,j,k)$
$u(i,j,k)$	$(j = k) \wedge (i = k)$	$a(i,j,k-1)$	$a(i,j,k-1)$
	$(j = k) \wedge (i > k)$	$u(i-1,j,k)$	$u(i-1,j,k)$
$b(i,j,k)$	$(j = k) \wedge (i = k)$		$[a(i,j,k-1)]^{-1}$
	$(j = k) \wedge (i > k)$		$b(i-1,j,k)$

Table 2: Program optimization by reducing common expensive operations.

5. Mapping Processes to Processors

System (4.1) derived above may be further improved by incorporating pipelining. From the standpoint of implementation, a process v in a system of recursion equations will be mapped to some physical functional unit, or processor s during execution, and once the process is terminated, another process can be mapped to the same processor. In fact, such re-use of the resource is the essence of *pipelining*. We call each execution of a process by a processor an *invocation* of the processor. Let t be an index for labeling the invocations so that the processes executed in the same processor can be differentiated, and

let these invocations be labeled by strictly increasing non-negative integers. Then for a given implementation of a program, each process $\mathbf{v}$ has an alias $[s,t] \stackrel{\text{def}}{=} f(\mathbf{v})$, telling when (which invocation) and where (in which processor) it is executed.

The key to an efficient parallel implementation of an algorithm is to find an appropriate one-to-one function f that maps a process $\mathbf{v}$ to its alias $[s,t]$ such that t will be non-negative and $t_2 > t_1$ if $\mathbf{v}_1 \prec \mathbf{v}_2$, where $[s_1,t_1] \stackrel{\text{def}}{=} f(\mathbf{v}_1)$ and $[s_2,t_2] \stackrel{\text{def}}{=} f(\mathbf{v}_2)$. Due to the page limitation, the method of *space-time mapping* is not presented here. Please refer to presentation elsewhere [5]. For LU decomposition, the method yields four different linear mapping functions, and three of them (denoted by T) are listed in Table 3.

dependency vec.	comm. vec. I	comm. vec. II	comm. vec. III
$\mathbf{d}_1 \stackrel{\text{def}}{=} (1,0,0)$	$[1,0,1]$	$\mathbf{c}_1 \stackrel{\text{def}}{=} [1,0,1]$	$[1,0,1]$
$\mathbf{d}_2 \stackrel{\text{def}}{=} (0,1,0)$	$[0,1,1]$	$\mathbf{c}_2 \stackrel{\text{def}}{=} [0,1,1]$	$[0,1,1]$
$\mathbf{d}_1 \stackrel{\text{def}}{=} (0,0,1)$	$[0,0,1]$	$\mathbf{c}_3 \stackrel{\text{def}}{=} [1,1,1]$	$[-1,-1,1]$
T	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$
	Algorithm I	Algorithm II	Algorithm III
data flow	[9, 8]	[3]	[11]

Table 3: Data dependency vectors, communication vectors, linear transforms *, and the data flows of systolic algorithms for LU decomposition.

6. Target Programs and Architectures

Proposition 6.1. *A space-time mapping* $T \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ *(Algorithm II) maps every process* (i,j,k) *(written as a column vector) of* P *to an invocation* (x,y,t) *(written as a column vector), or in functional notation:*

$$(x,y,t) = f(i,j,k) = (i+k, j+k, i+j+k)$$

where $1 \le x \le 2n$, $1 \le y \le 2n$, $(n-1) \le x-y \le n-1$ *and* $2 \le t \le 3n$. *The inverse mapping from the image of* f *to the domain of processes, denoted by* f^{-1} *is*

$$(i,j,k) = f^{-1}(x,y,t) = (t-y, t-x, x+y-t),$$

specifying at a particular invocation of a processor, which corresponding process is being executed.

The space-time mapping results in a new coordinate system with x-axis, y-axis, and t-axis as shown in Figure 2. Each process (i,j,k) is now mapped to some processor (x,y) at time step t. The resulting systolic algorithm with pipelining builtis obtained algebraically by substituting into System (4.1) the inverse mapping f^{-1} of i, j, and k. The program again is a system of recursion equations in **Crystal**, called *space-time recursion equations*, to indicate that each of the indices in the equations has a space or time interpretation. As described in Section 2, the system of equations has a parallel interpretation from which the local processing functions and communcation functions can be extracted from the algorithm. Once such factorization is made, the equations are ready to be translated to multiprocessor code or architectural-level specifications required by a silicon compiler. The formal algebraic transformations to obtain the space-time recursion equations are omitted here; rather, a few distinguishing features of the target program are discussed and shown in Figure 3.

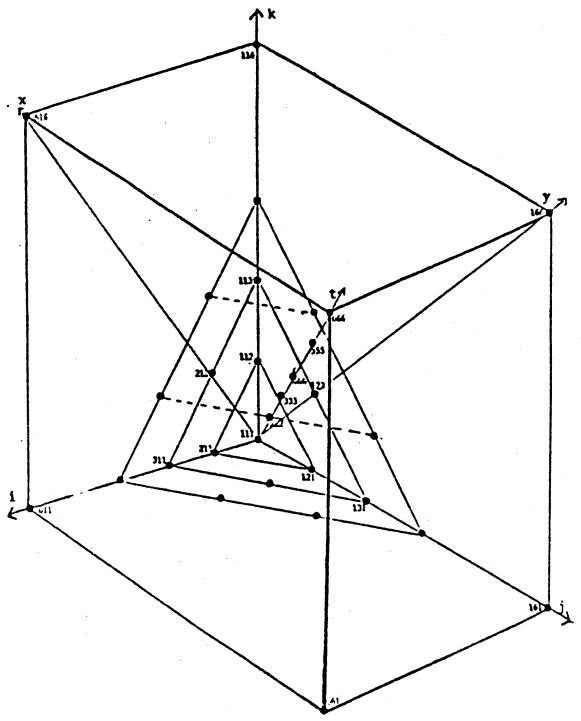

Figure 2: Processes in the original coordinate system and the one of Algorithm II. Processes on the same plane in any of the series of parallel planes can be executed at the same time step.

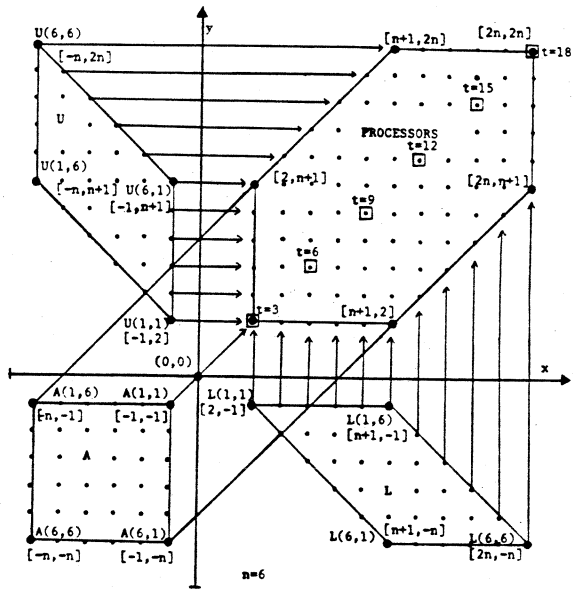

Figure 3: Processors and intial data streams of Algorithm II. Each of the processors that performs a division at some given time step is enclosed in a box.

6.1. Processor utilization

To find out if each processor (x,y) is active at every time step t, the inverse space-time mapping f^{-1} is used in obtaining the following two propositions.

Proposition 6.2. *Except for the boundary cases, a processor (x, y) which executes process (i, j, k) at time t will execute $(i+1, j+1, k-1)$ at time $t+1$ in Algorithm II.*

Proposition 6.3. *Except for the boundary cases, a processor (x, y) which executes process (i, j, k) at time t will execute $(i+1, j+1, k+1)$ at time $t+3$ in Algorithm III, but no process is executed at $(x, y, t+1)$ or $(x, y, t+2)$.*

By comparison, each processor is busy at every time step in Algorithm II [3] but only busy one out of every three time steps in Algorithm II [11].

6.2. Processor that performs division

From System (4.1), a process (i, j, k) performs division if $i = j = k$. Using the space-time mappings f:

Proposition 6.4. *An invocation (x, y, t) performs division if $t = \frac{3}{2}x = \frac{3}{2}y$ in Algorithm II; it performs division if $x = y = 0$ in Algorithm III.*

The control needed to indicate that a division $a(i, j, k-1)^{-1}$ should be performed is simple in the case of Algorithm III because $x = y = 0$ is a time-invariant expression, and hence is fixed statically. For Algorithm II, control code must be compiled into the object code to perform the test of the expression $t = \frac{3}{2}x = \frac{3}{2}y$. For a VLSI implementation of this algorithm, computing such control information locally at each processor is far too expensive to be desirable. Program optimization at the target program level that replaces time-variant control with time-invariant ones to avoid such computation is needed.

6.3. Target program optimizations

A better design can be obtained by replacing the expensive computations of a time-variant predicate by transferring a control signal. For a predicate that is independent of t, there is no concern, since it can be hardwired into the design. For any predicate that is dependent on t, it must be substituted by one that is independent of t. Since a communication always both moves in space and takes time to complete, it can be used to "compute" expressions of the space-time indices in a time-variant predicate. In the following, the control signal for the division operation in the new LU decomposition algorithm is derived. Please refer to [6] for the general method in deriving control signals.

The predicates $t < \frac{3}{2}x$, $t = \frac{3}{2}x$, and $t > \frac{3}{2}x$ mentioned above can be implemented by predicates $q(x, t) = 0$, $q(x, t) = 1$, and $q(x, t) = 2$, respectively, where the control stream is defined as

$$q(x,t) = \begin{cases} \frac{2}{3}x = 0 \rightarrow \begin{cases} t < 0 \rightarrow 0 \\ t = 0 \rightarrow 1 \\ t > 0 \rightarrow 2 \end{cases} \\ \frac{3}{2}x > 0 \rightarrow q(x-1, t-\frac{3}{2}) \end{cases}$$

This control stream can be easily implemented by a two-bit signal which moves "two processors every three time steps." Initially, its value is set to 1 and fed to processor $x = 0$ and then changes its value to 2 for $t > 0$, and shifts subsequently to processors with increasing x values one at a time. Readers might be curious about how, in general, the initialization of control stream for the *switch-on* predicates $t < 0$ is carried out in practice. The implementation is just the familar "system reset" which puts a given system into a desired initial state.

Central to an optimizing compiler for parallel systems is the making of trade-offs between communications and computations, i.e., trading local computation with global control signals or vice versa. Such trade-offs can be systematically devised and carried out by symbolic transformations in quite an elegant fashion, as illustrated here.

7. Concluding Remarks

The objective of this work is to demonstrate that it is possible to start with a high-level problem definition, and, by successive program transformations each aimed at optimizing the utilization of the underlying technologies, obtain a set of efficient parallel algorithms or architectures. The merits of these algorithms and architectures can be systematically compared and chosen for specific purposes. Due to the complexity that arises in dealing with hundreds of thousands of autonomous parallel processing elements, we find such a methodology and its automation highly desirable.

Automation of algorithm synthesis relies, at its most basic level, upon a formal notation for describing the problem, and henceforth upon manipulating the descriptions to yield efficient parallel algorithms. The essential nature of **Crystal** is that it is amenable to algebraic manipulation and is a general purpose programming language which allows new design methods and synthesis techniques, properties and theorems about problems in specific application domains, and new insights into any given problem be integrated within the existing program transformation and architecure simulation environment. Furthermore, all program transformations are carried out in **Crystal**, whether the issue is algorithmic or a concern of implementation. This capability of **Crystal** makes the tool that supports them portable from one multiprocessor environment to another, or to an entirely different medium, such as direct VLSI implementations.

References

[1] Chen, M. C., The Generation of a Class of Multipliers: a Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, October 1985, pp. 116–121.

[2] ———, *A Synthesis Method for Systolic Designs*, Technical Report 334, Yale University, January 1985.

[3] ———, Synthesizing Systolic Designs, *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, May 1985, pp. 209–215.

[4] ———, A Parallel Language and Its Compilation to Multiprocessor Machines, *The Proceedings of the 13th Annual Symposium on POPL*, January 1986, pp. 131–139.

[5] ———, Synthesizing VLSI Architectures: Dynamic Programming Solver, *The Proceedings of the 1986 International Conference on Parallel Processing*,, August 1986.

[6] ———, *A Design Methodology for Synthesizing Parallel Algorithms and Architectures*, Journal of Parallel and Distributed Computing, to appear (1986).

[7] Delosme J-M and Ipsen, Ilse, An illustration of a methodology for the construction of efficient systolic architecture in VLSI, *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, May 1985, pp. 268–273.

[8] Delosme, J. and Morf M., Scattering Arrays for Matrix Computations, *Proceedings of SPIE, Real time signal processing IV*, August 1981, pp. 74–91.

[9] Gentleman, W.M. and Kung, H.T., Matrix triangularization by Systolic Arrays, *Proceedings of SPIE, Real time signal processing IV*, August 1981, pp. 19–26.

[10] Kung, H.T., *Why Systolic Architecture?*, IEEE Computer, January (1982), pp. 37–46.

[11] Kung, H. T. and Leiserson, C. E., Algorithms for VLSI Processor Arrays, *Introduction to VLSI Systems by Mead and Conway*, Addison-Wesley, 1980.

[12] Li, G.-J. and Wah B. W., *The Design of Optimal Systolic Arrays*, IEEE Transactions on Computer, C-34/1 January (1985), pp. 66–77.

[13] Miranker, W. L., Spacetime Representations of Computational Structures, *Computing*, 1984, pp. 93–114.

[14] Moldovan, Dan I., On the Design of Algorithms for VLSI Systolic Arrays, *IEEE Transaction on Computer*, 1983.

[15] Moldovan, Dan. I., *ADVIS: A Software Package for the Design of Systolic Arrays*, Proceedings of ICCD, (1984).

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

References

[1] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, to appear 1986.

[2] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[3] M. C. Chen. Placement and interconnection of systolic processing elements: a new lu decomposition algorithm. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, page , October 1986.

[4] Jr. Halstead, Robert H. Multilisp: a language for concurrent symbolic computation. *ACM Transaction on Programming Language and Systems*, October 1985.

[5] C.A.R. Hoare. Communicating sequential processes. *Communication of ACM*, 21(8):666–677, 1978.

[6] Ming-Deh Huang. Solving some graph problems with optimal or near-optimal speedup on mesh-of-trees networks. In *Proceedings of the 26 Annual Symposium on Foundations of Computer Science*, pages 232–240, October 1985.

[7] Paul Hudak. Para-functional programming: a paradigm for programming multiprocessor systems. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[8] Richard M. Karp. *The Complexity of Parallel Computation*, pages 197–199. The MIT Press, Cambridge, Massachusetts, 1986.

[9] Charles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel graph algorithms. In *Proceedings of the 1986 International Conference on Parallel Processing*, page , 1986.

[10] Zhijung Mu and M. C. Chen. *Distributed Data Stuctures for Efficient Parallel Computations.* Technical Report in preparation, Yale University,, .

[11] Jeffrey D. Ullman. *COmputational Aspects of VLSI.* Computer Science Press, 1984.