In this paper, a design methodology for synthesizing efficient parallel algorithms and VLSI architectures is presented. A design process starts with a problem definition specified in the language **Crystal** and is followed by a series of program transformations in **Crystal**, each aiming at optimizing the target design for a specific purpose. To illustrate the design methodology, a set of design methods for deriving systolic algorithms and architectures is given and the use of these methods in the design of a dynamic programming solver is described. The design methodology, together with this particular set of design methods, can be viewed as a general theory of systolic designs (or multi-dimensional pipelines). The fact that **Crystal** is a general purpose language for parallel programming allows new design methods and synthesis techniques, properties and theorems about problems in specific application domains, and new insights into any given problem to be integrated readily within the existing design framework.

Yale University
Department of Computer Science

A Design Methodology for Synthesizing Parallel Algorithms and Architectures

Marina C. Chen

YALEU/DCS/TR-457
June 1986

Work supported by ONR Contract N00014-86-K-0296. Approved for public release: distribution is unlimited

[1]To appear in Journal of Parallel and Distributed Computing.

Table of Contents

A Design Methodology for Synthesizing Parallel Algorithms and Architectures

Marina C. Chen
Department of Computer Science, Yale University
New Haven, CT 06520

Abstract In this paper, a design methodology for synthesizing efficient parallel algorithms and VLSI architectures is presented. A design process starts with a problem definition specified in the parallel programming language **Crystal** and is followed by a series of program transformations in **Crystal**, each aiming at optimizing the target design for a specific purpose. To illustrate the design methodology, a set of design methods for deriving systolic algorithms and architectures is given and the use of these methods in the design of a dynamic programming solver is described. The design methodology, together with this particular set of design methods, can be viewed as a general theory of systolic designs (or multi-dimensional pipelines). The fact that **Crystal** is a general purpose language for parallel programming 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 readily within the existing design framework.

1. Introduction

VLSI technology provides a natural medium for parallel processing, from the level of switching elements, to logic gates, to functional and control units, and to architectures and algorithms. To expoit its potential fully, parallelism must be used at increasingly higher levels. Design automation, consequently, must go beyond the level of taking as givens architectural level specifications, and should head towards compiling or synthesizing efficient parallel algorithms and architectures. 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 the design.

Motivated by the former, a *design methodology* for synthesizing efficient parallel algorithms and VLSI architectures is presented. A design process starts with a problem definition specified in the parallel programming language **Crystal** [4] and is followed by a series of program transformations in **Crystal**, each aiming at optimizing the target design for a specific purpose. To illustrate the design methodology, a set of *design methods* for deriving systolic algorithms and VLSI architectures, motivated by the costs and constraints of the technology, is given. The use of these methods in the design of a dynamic programming solver is described. The design methodology, together with this particular set of design methods, can be viewed as a general theory of systolic designs (or multi-dimensional pipelines).

Each design method or synthesis technique takes into account the dominant costs in the underlying technology and minimizes such costs to yield efficient parallel algorithms and architectures. For instance, the design process starts with a problem definition specified in **Crystal**, which is interpreted as a parallel algorithm, however naive and inefficient it might be. From this naive algorithm, the dominating costs it might incur in the underlying parallel hardware, such as those of communications and fan-ins and fan-outs, are examined. The algorithm is then improved by a series of transformations that results in a *bounded order* and *bounded degree* program which has reduced hardware costs. Once such a program is obtained, it goes through another stage of transformation, called *space-time mapping*, by which pipelining is automatically incorporated into the algorithm and the hardware resources are fully utilized. Another stage of transformations then follows, which aims at reducing the amount of hardware necessary for achieving the timing control among the parallel processing elements.

The set of synthesis methods for the design of VLSI architecture described in this paper has a wide range of applications, but is by no means a complete set of general synthesis methods. In fact, it is hard to believe such a set may ever be attainable. However, the fact that **Crystal** is a general purpose parallel programming language allows new synthesis techniques, new theorem, and new insights for any give problem be integrated readily within the existing design framework.

Throughout this paper, the dynamic programming problem is used for illustration. The well-known systolic algorithm of Guibas, Kung, and Thompson [7] is one of the many possible designs generated. The rest of the paper is organized as follows: In Section 2, a mathematical definition of dynamic programming is given. In Section 3, its interpretation as a parallel algorithm is described. In Section 4, transformations for reducing the number of fan-ins and fan-outs are presented. In Section 5, transformations that reduce long-range communications are illustrated. In Section 6, algorithms are classified as either uniform or non-uniform because space-time mapping for uniform algorithms can be obtained by an expedient procedure, which is described here. A general inductive method for finding space-time mapping for non-uniform algorithms can be found in [5]. In Section 7, optimization of control signals is illustrated. Some related work is discussed in Section 8. The appendix contains the actual **Crystal** programs, starting with a simple mathematical definition in the first program to a complete architectural specification in the target program. An architecutral level simulation of the design is generated as a by-product of executing the target program. Therefore the **Crystal** programming environment automatically provides a simulation environment for the design of VLSI architectures.

2. A General Definition of Dynamic Programming

A large class of optimization problems can be solved by dynamic programming. The definition of such problems can be posed in a general form where $C(i,j)$ is some cost function which is to be minimized:

$$C(i,j) = \begin{cases} s > 0 \rightarrow \min_{i<k<j} \{h(C(i,k), C(k,j))\} \\ \qquad \text{where } h \text{ is some function on the costs} \\ s = 0 \rightarrow C_i \text{ for some individual cost} \end{cases} \qquad (2.1)$$

where $s = j - i - 1$,

and i and j are integers in the range $0 < i < j \leq n$, for some constant n.

3. Parallel Interpretation of Algorithms

The straightforward definition given in Equation (2.1) is in the form of a **Crystal** recursion equation with indices i and j. It can be interpreted as a parallel algorithm as follows: Each pair (i,j) in Equation (2.1) is interpreted as a process in an ensemble of parallel processes denoted by the set $P_1 \stackrel{\text{def}}{=} \{(i,j) : 0 < i < j \leq n\}$. The local processing at each process (i,j) is, in this example, the function $\min_{i<k<j}\{h(x_k, y_k)\}$ that takes $j - i - 1$ pairs of arguments. The communication between processes is specified, for instance, by the pairs (i,k) and (k,j) appearing on the right-hand side of Equation (2.1) to indicate that the computation of $C(i,j)$ at process (i,j) needs the results from processes (i,k) and (k,j).

The ensemble of processes and its data flow can be depicted by a DAG (Directed Acyclic Graph) as shown in Figure 1. It consists of nodes, where each node corresponds to a process (i,j) in P_1, and directed edges, each of which comes out of a node whose corresponding process appears on the right-hand side of Equation (2.1), and goes into a node whose corresponding process appears on the left-hand side of the equation. The directed edges of the DAG define the data dependency relation of the algorithm. We say that a process **u** immediately precedes **v** ($\mathbf{u} \prec \mathbf{v}$), or **v** immediately depends on **u** ($\mathbf{v} \succ \mathbf{u}$), if there is a directed edge from **u** to **v**. The transitive closure "$\stackrel{*}{\prec}$" of this relation, called "precede", 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**. Those nodes that have no incoming edges are called *sources*.

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. The parallel execution of the naive dynamic programming definition starts with all processes such that $s = 0$, followed by processes with increasing s as illustrated in Figure 1.

An immediate, very naive implementation of the parallel interpretation uses one processor for each process, and altogether $O(n^2)$ parallel processors are needed. The number of time steps needed to compute the result is at best $n-1$ since any $C(i,j)$ cannot be computed until $C(i,j-1)$ and $C(i+1,j)$ are computed.

4. Problems with Large Numbers of Fan-ins and Fan-outs

Due to the inherent physical constraints imposed by the driving capability of communication channels,

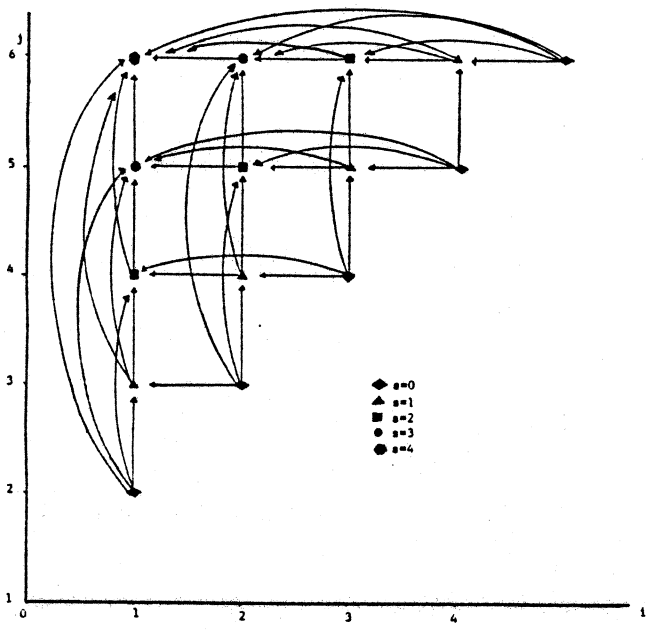

Figure 1: The DAG describing the data dependency of dynamic programming.

power consumptions, heat dissipations, memory bandwidth, etc., it is reasonable to assume that data can only be sent or received simultaneously to or from a small number of destinations or sources. Putting such constraints in algorithmic terms: the number of "fan-ins" and "fan-outs" of a datum must be small, where *fan-in degree* of a datum is defined to be the number of data items on which it depends, and *fan-out degree* is the number of data items dependent upon it. If what we are interested in are practical and efficient algorithms, the fan-in and fan-out degrees should be taken into account in measuring the complexity.

It can be easily seen that the fan-in degree of $C(i,j)$, which appears on the left-hand side of Equation (2.1), is $2s$, where $s \stackrel{\text{def}}{=} j - i - 1$. Conversely, for any value appearing as $C(i,k)$ on the right-hand side of Equation (2.1), that value is needed by $C(i,j)$ for $j = k+1, \ldots, n$ (there are $n-k$ such terms). The same value also appears as $C(k,j)$ in the equation and it is needed by processes (i,j) for $i = 1, \ldots, k-1$ (there are $k-1$ such terms). Since both s and k grow with the problem size n, the number of fan-ins and fan-outs therefore grows linearly with the size of the problem.

To avoid the high cost incurred at the hardware level, the large fan-in and fan-out degrees must be reduced. In the following transformations, the original fan-in and fan-out degrees are reduced from $O(n)$ to a constant, while the complexity of the original algorithm is maintained the same as before.

4.1. Reducing fan-out degrees

The idea used in reducing the fan-out degree is quite simple:

Proposition 4.1. *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*

$$z(l) = \begin{cases} l = u \rightarrow x \\ \\ l > u \rightarrow z(l-1) \end{cases} \qquad (4.1)$$

or by

$$z(l) = \begin{cases} l = v \rightarrow x \\ \\ l < v \rightarrow z(l+1) \end{cases} \qquad (4.2)$$

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 $z(l)$, for all $u \leq l \leq v$. 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 broadcasting 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.

For the dynamic programming problem, in order to decrease the fan-out degree of $C(i,k)$, the schemes of linear broadcasting array and the broadcast tree are first compared. The former is chosen since the time complexity of the target program employing each scheme turns out to be of the same order for both cases, and the linear broadcasting array is simpler and requires less hardware than a broadcasting tree. Within the class of linear broadcasting 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.

Now let the variables at processes (i,j) for $k \leq j \leq n$ to which $C(i,k)$ will be assigned be denoted by $a(i,j,k)$ so that Proposition 4.1 can be applied to $C(i,k)$ (the value x) and $a(i,j,k)$ for all j (the variables $z(l)$). Similarly, let the variables at process (i,j) for $1 \leq k < i$ to which $C(k,j)$ is to be assigned be denoted

by $b(i,j,k)$. Applying Equation (4.1) of the proposition to $a(i,j,k)$ and Equation (4.2) to $b(i,j,k)$, we obtain

$$a(i,j,k) = \begin{cases} j = k \rightarrow C(i,k) \\ j > k \rightarrow a(i,j-1,k) \end{cases}$$
$$b(i,j,k) = \begin{cases} i = k \rightarrow C(k,j) \\ i < k \rightarrow b(i+1,j,k) \end{cases} \qquad (4.3)$$
$$\text{for } 1 \leq i < k < j \leq n.$$

Now the $n-k$ fan-out degree of $C(i,k)$ to all processes (i,j), where $k < j \leq n$, is reduced to 1, and so is the $k-1$ fan-out degree of $C(k,j)$ to all processes (i,j) for $1 \leq i < k$. We may now replace the high fan-out degree values $C(i,k)$ and $C(k,j)$ on the right-hand side of Equation (2.1) by the low fan-out degree values $a(i,j,k)$ and $b(i,j,k)$:

$$C(i,j) = \begin{cases} s > 0 \rightarrow \min\limits_{i<k<j} \{h(a(i,j,k), b(i,j,k))\} \\ s = 0 \rightarrow C_i. \end{cases} \qquad (4.4)$$

4.2. Reducing the fan-in degrees

The reduction of fan-in degree of a value computed by a process relies on the associativity of the operation being computed.

Definition 4.1. An operation "$\bigoplus$", where $z = \bigoplus_{u<l<v} x(l)$, on $v-u-1$ arguments, is associative if it can be replaced by the composition of the following sequence of binary associative operations "$\oplus$" on variables $z(l)$, $u < l < v$:

$$z(l) = \begin{cases} l = u+1 \rightarrow x(u+1) \\ u+1 < l < v \rightarrow z(l-1) \oplus x(l) \end{cases} \qquad (4.5)$$

and $z = z(v-1)$.

Similarly, the same operation can be achieved by a different sequence using both binary and ternary associative operations.

Proposition 4.2. *If "$\bigoplus$" is an associative operation on $v-u-1$ arguments $x(l)$ for $u < l < v$, where $z = \bigoplus_{u<l<v} x(l)$, then it can be replaced by a sequence of binary and ternary operations on variables $z(l)$, for $m < l \leq v$ of the following form:*

$$z(l) = \begin{cases} l = m \rightarrow x(m) \oplus x(u+v-m) \\ m < l < v \rightarrow z(l-1) \oplus x(l) \oplus x(u+v-l) \\ l = v \rightarrow z(l-1) \end{cases} \qquad (4.6)$$
$$\text{where } m = \lceil \frac{v+u}{2} \rceil$$

and $z = z(v)$.

Similar to the case of reducing fan-out degrees, there could be many ways that fan-in degrees can be reduced. Aside from the two linear schemes described by the recursion equations above, a merge tree which

takes a logarithmic number of steps to complete an n-ary operation can be used. Again, the choice of these two very different schemes is problem dependent. Capturing these schemes by recursion equations in a formal programming language allows the chosen transformation be performed mechanically, and the various choices be made either manually by the users or, with the help of a set of *ad hoc* rules for making choices, automatically by a design tool.

For dynamic programming, note that the operation "min" for computing $C(i,j)$ is associative. Therefore we may let the high fan-in degree value $C(i,j)$ at process (i,j) be computed instead by the sequence $c(i,j,k)$ for $m < k < j$, where $m \stackrel{\text{def}}{=} \lceil \frac{i+j}{2} \rceil$. We apply Proposition 4.2 to Equation (4.4), and it is transformed to

$$c(i,j,k) = \begin{cases} s = 0 \rightarrow C_i \\ s > 0 \rightarrow \\ \quad \begin{cases} k = m \rightarrow \min(h(a(i,j,k), b(i,j,k)), \\ \qquad\qquad h(a(i,j,i+j-k), b(i,j,i+j-k)) \\ m < k < j \rightarrow \\ \quad \min(c(i,j,k-1), h(a(i,j,k), b(i,j,k)), \\ \qquad\qquad h(a(i,j,i+j-k), b(i,j,i+j-k)) \\ k = j \rightarrow c(i,j,k-1), \end{cases} \end{cases} \tag{4.7}$$

and $C(i,j) = c(i,j,j)$ for all $(i,j) \in P_1$.

The application of Proposition 4.2 is motivated by the following: Due to the data dependency, the value $h(a(i,j,k), b(i,j,k))$ for $i < k < j$ cannot be computed before at least $\max(k-i, j-k)$ time steps. A pair with the least delay is the one with $k = \lfloor \frac{i+j}{2} \rfloor$ and/or $k = \lceil \frac{i+j}{2} \rceil$, where the delay is about half of the interval size. Consequently, the most efficient sequence of binary or ternary operations will start with this pair, and will be followed by other pairs in the order of increasing amount of delay.

5. Communcations and Bounded-order Recursion Equations

The following two equations

$$\begin{aligned} a(i,j,k) &= \begin{cases} j = k \rightarrow c(i,k,k) \\ j > k \rightarrow a(i,j-1,k) \end{cases} \\ b(i,j,k) &= \begin{cases} i = k \rightarrow c(k,j,j) \\ i < k \rightarrow b(i+1,j,k) \end{cases} \\ &\text{for } 1 \le i < k < j \le n, \end{aligned} \tag{5.1}$$

are obtained from Equation (4.3) by substituting $C(i,k)$ with $c(i,k,k)$, and substituting $C(k,j)$ with $c(k,j,j)$. The system of recursion equations consisting of Equation (5.1) and Equation (4.7) is a new algorithm for dynamic programming. The new ensemble of processes $P_2 \stackrel{\text{def}}{=} \{(i,j,k) : 0 < i < k < j \le n\}$ is now three-dimensional, as shown in Figure 2. The added one dimension and the increased number of

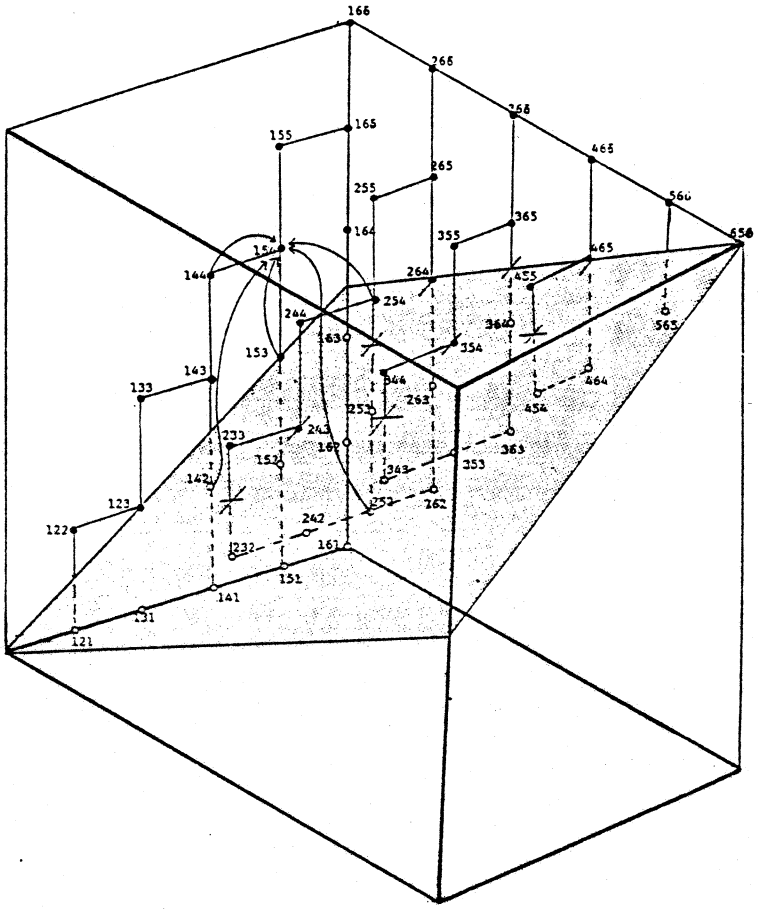

Figure 2: Process structure P_2 in which the fan-in and fan-out degreess are constant, but in which there are still long range communications.

processes are for the purpose of decreasing fan-in/fan-out degrees. A remaining criterion for a good parallel algorithm is that all communications between processes should be as local as possible, if such localization is not achieved at the expense of the total time complexity of the algorithm. To define locality, a few definitions are called for:

Definition 5.1. The path length between two processes $\mathbf{v} \stackrel{\text{def}}{=} (v_1, \ldots, v_m)$ and $\mathbf{u} \stackrel{\text{def}}{=} (u_1, \ldots, u_m)$ is defined to be $|v_1 - u_1| + \cdots + |v_m - u_m|$.

Definition 5.2. A communication between two processes $\mathbf{u}$ and $\mathbf{v}$, where $\mathbf{u} \prec \mathbf{v}$, is local if their path length is bounded by a fixed constant.

Definition 5.3. The *order* of a system of recursion equations is defined to be the maximum path length between any pair of processes $\mathbf{u}$ and $\mathbf{v}$, where $\mathbf{u} \prec \mathbf{v}$.

Similar to the high fan-in and fan-out degrees, the extra delay introduced by the communications of the high-order terms in the equations can undermine the efficiency of an algorithm. Therefore the order of

a system of equations should be either constant or grow very slowly with respect to the problem size.

5.1. Localizing communications by domain contraction

Note that Equation (4.7) contains terms $a(i,j,i+j-k)$ and $b(i,i+j-k,j)$, and the order of the system of equations is therefore $2(|k-\frac{i+j}{2}|)$ for $i<k<j$, which grows with the problem size n. To eliminate the high order terms, further transformations are required.

The path length $2(|k-\frac{i+j}{2}|)$ between two communicating processes is an expression whose value depending on those of i, j and k. The goal is to transform this expression into one whose value is bounded. The most obvious choice is to set $k=\frac{i+j}{2}$, in which case the path length becomes 0, i.e., two processes are collapsed into a single one. Geometrically, the coordinates of these high order terms are symmetric to those of the low-order terms $a(i,j,k)$ and $b(i,j,k)$, with respect to the plane $k=(i+j)/2$ which is shaded as shown in Figure 2. The algebraic transformation now corresponds to the contraction of the original domain of processes by one half along the plane, so that (i,j,k) and $(i,j,i+j-k)$ become a single process (i,j,k).

Let processes (i,j,k), such that $m \le k \le j$, be referred to as being in the upper half of the process structure P_2. Similarly, let those processes, such that $i \le k \le m'$, be referred to as being in the lower half of P_2.

Let the new process structure be the upper half of P_2, i.e., $P_3 \stackrel{\text{def}}{=} \{(i,j,k) : 0<i<j\le n \text{ and } m \le k \le j\}$, as shown in Figure 3. The data streams a and b of those processes originally residing in the lower half must be given new names so that they can be differentiated from those of the processes originally in the upper half. To achieve this, let d and e be new data streams to replace a and b for the processes of the lower half as shown in Figure 3.

5.2. Algebraic manipulation to obtain bounded-order equations

First, Equation (5.1), defined for (i,j,k), $i \le k \le j$, is now split into two sets of equations, one for the case $m \stackrel{\text{def}}{=} \lceil \frac{i+j}{2} \rceil \le k < j$ where (i,j,k) resides in the upper half of P_2, and the other for the case $i<k\le m' \stackrel{\text{def}}{=} \lfloor \frac{i+j}{2} \rfloor$ where (i,j,k) resides in the lower half. Since, for those processes (i,j,k) that reside in the lower half, a and b will be renamed by d and e, we must be aware of the case when a process (i,j,k) is in the lower half of P_2 while process $(i,j-1,k)$ might be in the upper half, such as in the case when $k=m'$ and $i+j$ is even (or $s \stackrel{\text{def}}{=} j-i-1$ is odd). Another case to watch for is when (i,j,k) is in the upper half while $(i+1,j,k)$ might be in the lower half: it occurs when $k=m$ and $i+j$ is even (or s is odd). Two

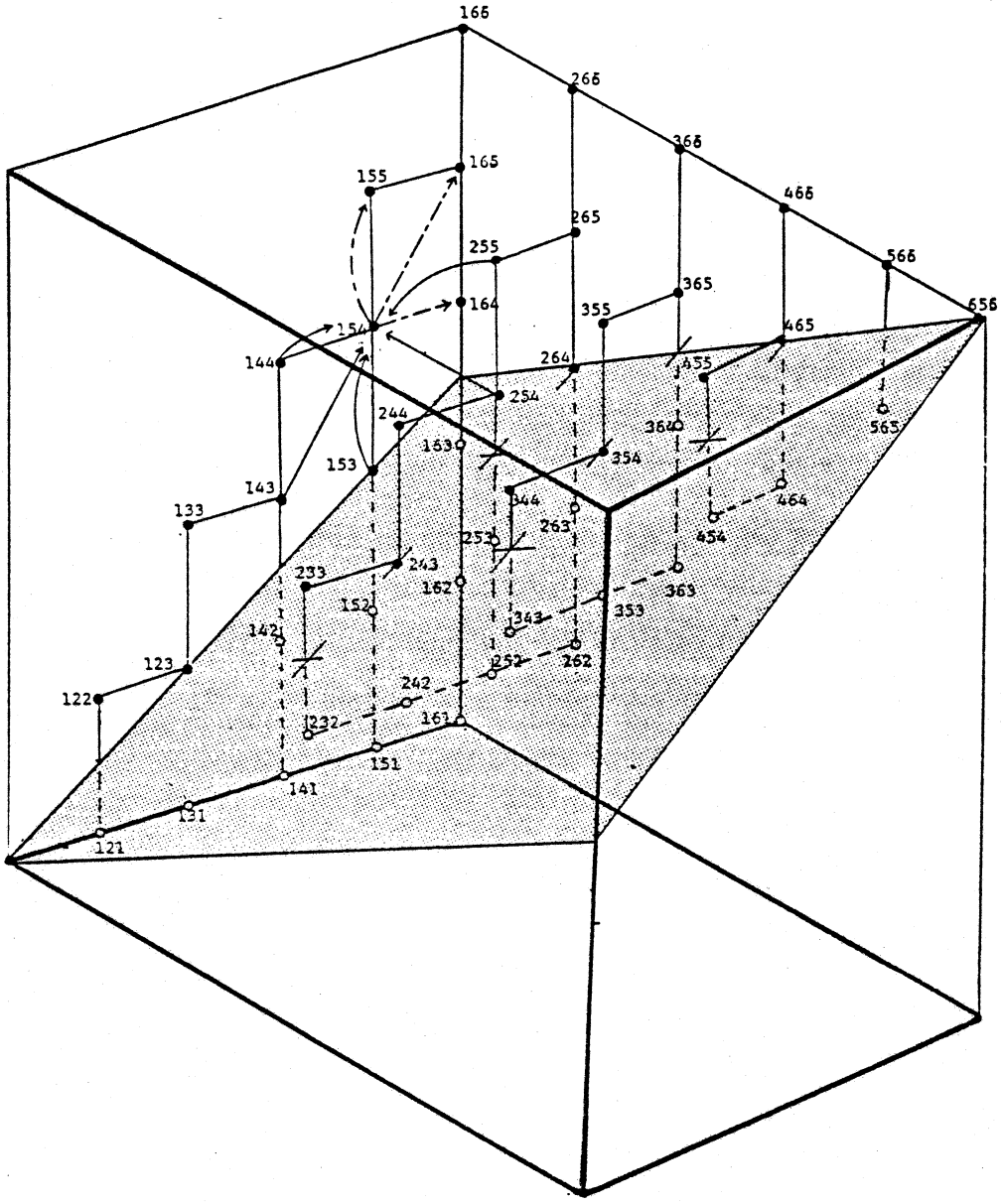

Figure 3: Process structure P_3, in which all data dependencies are local.

equations in (5.1) are now split into four equations, where the above two cases are singled out:

$$
\begin{aligned}
&\text{upper half:}\\
&a(i,j,k) = \begin{cases} j = k \rightarrow c(i,k,k) \\ m \leq k < j \rightarrow a(i,j-1,k). \end{cases}\\
&\text{lower half:}\\
&a(i,j,k) = \begin{cases} (k = m') \wedge (odd(s)) \rightarrow a(i,j-1,k) \\ \qquad \text{(upper half)} \\ (k = m') \wedge (even(s)) \rightarrow a(i,j-1,k) \\ i < k < m' \rightarrow a(i,j-1,k). \end{cases}\\
&\text{upper half:}\\
&b(i,j,k) = \begin{cases} (k = m) \wedge (odd(s)) \rightarrow b(i+1,j,k) \\ \qquad \text{(lower half)} \\ (k = m) \wedge (even(s)) \rightarrow b(i+1,j,k) \\ m < k < j \rightarrow b(i+1,j,k). \end{cases}\\
&\text{lower half:}\\
&b(i,j,k) = \begin{cases} k = i \rightarrow c(k,j,j) \\ i < k \leq m' \rightarrow b(i+1,j,k). \end{cases}
\end{aligned}
\tag{5.2}
$$

Next, every $a(i,j,k)$ in the lower half is replaced by $d(i,j,i+j-k)$, where $(i,j,i+j-k)$ is now the upper half. Similarly, every $b(i,j,k)$ in the lower half is replaced by $e(i,j,i+j-k)$:

$$
\begin{aligned}
&(i,j,k) \text{ lower half}\\
&d(i,j,i+j-k) = \begin{cases} (k = m') \wedge (odd(s)) \rightarrow a(i,j-1,k) \\ \qquad \text{(upper half)} \\ (k = m') \wedge (even(s)) \rightarrow \\ \quad d(i,j-1,i+(j-1)-k) \\ i < k < m' \rightarrow d(i,j-1,i+(j-1)-k). \end{cases}\\
&(i,j,k) \text{ lower half}\\
&e(i,j,i+j-k) = \begin{cases} k = i \rightarrow c(k,j,j) \\ i < k \leq m' \rightarrow e(i+1,j,(i+1)+j-k). \end{cases}
\end{aligned}
\tag{5.3}
$$

Finally, a "substitution of variable" is performed on the above two equations, which replaces $i+j-k$

by k':

$$d(i,j,k') = \begin{cases} (k' = m) \wedge (odd(s)) \rightarrow a(i, j-1, i+j-k') \\ (k' = m) \wedge (even(s)) \rightarrow d(i, j-1, k'-1) \\ m < k' < j \rightarrow d(i, j-1, k'-1). \end{cases}$$

$$e(i,j,k') = \begin{cases} k' = j \rightarrow c(i,j,j) \\ m \leq k' < j \rightarrow e(i+1, j, k'+1). \end{cases} \qquad (5.4)$$

Since k' is a bound variable, it can be replaced by k throughout Equations (5.4) without any effect. Note also that when $k' = m$, we have $i + j - k' = k'$.

5.3. Resulting system of equations

The resulting algorithm is the following system of recursion equations:

$$a(i,j,k) = \begin{cases} j = k \rightarrow c(i,k,k) \\ m \leq k < j \rightarrow a(i, j-1, k) \end{cases}$$

$$d(i,j,k) = \begin{cases} (k = m) \wedge (odd(s)) \rightarrow a(i, j-1, k) \\ (k = m) \wedge (even(s)) \rightarrow d(i, j-1, k-1) \\ m < k < j \rightarrow d(i, j-1, k-1) \end{cases}$$

$$b(i,j,k) = \begin{cases} (k = m) \wedge (odd(s)) \rightarrow e(i+1, j, k+1) \\ (k = m) \wedge (even(s)) \rightarrow b(i+1, j, k) \\ m < k < j \rightarrow b(i+1, j, k) \end{cases}$$

$$e(i,j,k) = \begin{cases} k = j \rightarrow c(i,j,j) \\ m \leq k < j \rightarrow e(i+1, j, k+1) \end{cases} \qquad (5.5)$$

$$c(i,j,k) = \begin{cases} s = 0 \rightarrow C_i \\ s > 0 \rightarrow \\ \quad \begin{cases} k = m \rightarrow \min(h(a(i,j,k), b(i,j,k)), \\ \qquad\qquad h(d(i,j,k), e(i,j,k)) \\ m < k < j \rightarrow \\ \quad \min(c(i,j,k-1), h(a(i,j,k), b(i,j,k)), \\ \qquad\qquad h(d(i,j,k), e(i,j,k)) \\ k = j \rightarrow c(i,j,k-1). \end{cases} \end{cases}$$

The order of the system of equations is now 2, due the path length between processes described by equations for streams b and d.

6. Space-time Mapping to Incorporate Pipelining

From the algorithm (5.5) derived above, we further improve the efficiency of the algorithm by incorporating pipelining automatically. From the standpoint of implementation, a process $\mathbf{v}$ in a system of recursion equations will be mapped to some physical 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 $[s_2,t_2] \stackrel{\text{def}}{=} f(\mathbf{v}_2)$. In the following , we call t the time index and each component of s a space index.

6.1. Data dependency vectors

To find such a mapping, we require that the domains of the indices of a **Crystal** program be vector spaces, and that the appropriate vector addition and scaler vector product be defined. Though each of the indices i and j in Equation (2.1) assumes an integer value, its domain can be extended to the set of rationals. For instance, an m dimensional process structure is now embedded in an m-dimensional vector space over the rationals. From now on, we may refer to the m-tuple of values of indices identified with a process as a vector. As suggested by [12], a data dependency vector plays an important role in mapping algorithms to parallel processors. Since the vector addition is defined and the data dependency of two vectors takes on an exact meaning, a data dependency vector can be formally defined:

Definition 6.1. A *data-dependency vector* is the difference $\mathbf{v} - \mathbf{u}$ of vector $\mathbf{v}$ and vector $\mathbf{u}$, where $\mathbf{u} \prec \mathbf{v}$ ($\mathbf{u}$ immediately precedes $\mathbf{v}$).

6.2. Basis communication vectors

As described above, each program defines a set of data dependency vectors. On the other hand, each network topology defines a set of linearly independent *basis communication vectors*. For instance, in an n-dimensional hypercube, a processor has n connections to its nearest neighboring processors. Each of the n communication vectors (one for each connection), has $n+1$ components. The first n components indicate the movement in space and the $(n+1)$th component indicates movement in time, which is always positive (counting invocations). These n communication vectors, together with the communication vector $[0,0,\ldots,0,1]$ representing the processors's communication of its current state to its next state, form the basis communication vectors. In an n-dimensional network, there can be more than one set of basis communication vectors. Taking a two-dimensional hexagonal network as an example, a diagonal connection has a communication vector $[1,1,1]$. The set of vectors $\{[1,0,1],[0,1,1],[1,1,1]\}$ serves as the bases as well as

the set $\{[1,0,1],[0,1,1],[0,0,1]\}$. For any n-dimensional network which has nearest neighbor connections and is regularly connected and indefinitely extensible, all possible sets of basis communication vectors can be obtained by the enumeration of its symmetry groups (Lin and Mead [10]).

6.3. Uniformity of a parallel algorithm

The concept of uniformity is introduced to characterize parallel algorithms so that an expedient procedure can be applied to a *uniform algorithm* to find the space-time mapping from processes to processors. Let $\mathbf{d}_i$, $1 \leq i \leq k$, denote the data dependency vectors appearing in a program.

Definition 6.2. A uniform algorithm is one in which a single set of basis vectors $B = (\mathbf{b}_1, \ldots, \mathbf{b}_m)$, $m \leq k$, where each $\mathbf{b}_j$ is a column vector of m components, can be chosen so as to satisfy the mapping condition that every data dependency vector $\mathbf{d}_i$ appearing in the algorithm can be expressed as a linear combination of the chosen basis vectors with non-negative coordinates, in other words, there exists $\mathbf{a}_i = \begin{pmatrix} \alpha_{i1} \\ \vdots \\ \alpha_{im} \end{pmatrix}$ such that $\mathbf{d}_i = B\mathbf{a}_i$ and $\alpha_{ij} \geq 0$ for $1 \leq j \leq m$.

6.4. Motivation for the mapping condition

The mapping condition is motivated by the possibility of using as the space-time mapping a linear transform from the basis data dependency vectors B to the basis communication vectors $C = (\mathbf{c}_1, \ldots, \mathbf{c}_m)$, $m \leq k$, where each $\mathbf{c}_j$ is a column vector of m components. Each basis communication vector $\mathbf{c}_j$ corresponds to a nearest neighbor communication on a network of processors. When the mapping between the two sets of basis vectors is determined, then each communication vector $\mathbf{e}_i$, $1 \leq i \leq k$, to which a data dependency vector $\mathbf{d}_i$ corresponds is also determined, i.e., if $\mathbf{d}_i = B\mathbf{a}_i$ then $\mathbf{e}_i = C\mathbf{a}_i$.

If a data dependency vector $\mathbf{d}_i$ has a term with a negative coordinate α_{ij} in its linear combination, then vector $\alpha_{ij}\mathbf{c}_j$, which contributes in part to the communication vector $\mathbf{e}_i$, represents a communication that takes negative time steps, a situation that does not make sense in any physical implementation.

Examples of using such a simple procedure to find linear mappings of processes to parallel architectures for matrix products, LU decomposition, array multipliers, etc., can be found in [2, 3]. Most of the systolic algorithms reported in the literature can be obtained this way. New systolic algorithms are in fact discovered due to the ability to generate systematically all possible sets of basis communication vectors. In the case of a non-uniform program, more than one set of basis data dependency vectors must be chosen so as to satisfy the mapping condition, and the space-time mapping may become non-linear. In this case, the inductive mapping procedure becomes necessary.

6.5. A uniform dynamic programming algorithm

Let $\mathbf{d}_i$, $i = 1, 2, \ldots, 5$ denotes the data dependency vectors of Systems (5.5):

$$\begin{aligned} &\mathbf{d}_1 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{d}_2 \stackrel{\text{def}}{=} \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}, \mathbf{d}_3 \stackrel{\text{def}}{=} \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}, \\ &\mathbf{d}_4 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \mathbf{d}_5 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \end{aligned} \tag{6.1}$$

Let the following vectors be the chosen basis data dependency vectors:

$$\mathbf{b}_1 \stackrel{\text{def}}{=} \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}, \mathbf{b}_2 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{b}_3 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

and let matrix B be defined as $B \stackrel{\text{def}}{=} (\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3)$, then

$$\begin{aligned} &\mathbf{d}_i = B\mathbf{a}_i, \text{ where} \\ &\mathbf{a}_1 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{a}_2 \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \mathbf{a}_3 \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \\ &\mathbf{a}_4 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \mathbf{a}_5 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \end{aligned}$$

Since every component α_{ij} of $\mathbf{a}_i$ is non-negative, thus System (5.5) of dynamic programming is a uniform algorithm. Choose the set of basis communication vectors as

$$\mathbf{c}_1 \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \mathbf{c}_2 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \mathbf{c}_3 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \text{ and } C \stackrel{\text{def}}{=} (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3). \tag{6.2}$$

The linear mapping T from the basis dependency vector to the basis communication vector is then

$$T = CB^{-1} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 1 & 1 \end{pmatrix}.$$

Proposition 6.1. *A space-time mapping from each process (i, j, k) in P_3 of System (5.5) to an invocation of a processor (x, y, t) is a linear mapping T, in the matrix notation where each process and invocation is written as a column vector, or written in a functional notation as*

$$\begin{aligned} &(x, y, t) = f(i, j, k) = (-i, j, -2i + j + k) \text{ where} \\ &-n < x \leq -1, \ -x < y \leq n, \ 2 \leq t < 2(n-1). \end{aligned}$$

The inverse mapping from the image of f to the process structure, denoted by f^{-1}, is

$$(i, j, k) = f^{-1}(x, y, t) = (-x, y, -2x - y + t),$$

specifying which process is being executed at a particular time step t of processor (x, y).

Similarly, other space-time mappings can be derived from other sets of basis communication vectors such as B_1, B_2, and B_3 given below in a matrix notation where each basis communication vector is written as a column vector:

$$B_1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \; B_2 = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}, \; B_3 = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}. \tag{6.3}$$

The set of basis communication vectors in Equation (6.2), or in each of (6.3) above, results in a different systolic architecture, as demonstrated below.

6.6. Space-time recursion equations (STREQ)

A system of *space-time recursion equations* (STREQ), which describes a target systolic architecture or algorithm, can be obtained from an original program and a space-time mapping by algebraic manipulation. The following STREQ that describe the resulting design are obtained by substituting the inverse mapping f^{-1} of i, j, and k, the mappings $f(\mathbf{d}_l)$, $l = 1, 2 \ldots, 5$ of dependency vectors $\mathbf{d}_l$, into the original system of recursion equations, and by renaming $\hat{a}(x, y, t) = a(i, j, k)$, $\hat{b}(x, y, t) = b(i, j, k)$, $\hat{c}(x, y, t) = c(i, j, k)$, $\hat{d}(x, y, t) = d(i, j, k)$, and $\hat{e}(x, y, t) = e(i, j, k)$. Note that as a result, the substitution of the predicates $s = 0$ becomes $z = 1$, where $z \stackrel{\text{def}}{=} x + y$, $k = m$ becomes $t = \lceil \frac{3(x+y)}{2} \rceil = \lceil \frac{3z}{2} \rceil$, $k = j$ becomes $t = 2z$, $m < k < j$ becomes $(\lceil \frac{3z}{2} \rceil < t < 2z)$, etc.

$$\hat{a}(x,y,t) = \begin{cases} t = 2z \rightarrow \hat{c}(x,y,t) \\ \lceil \frac{3z}{2} \rceil \leq t < 2z \rightarrow \hat{a}(x,y-1,t-1) \end{cases}$$

$$\hat{d}(x,y,t) = \begin{cases} (t = \lceil \frac{3z}{2} \rceil) \wedge (even(z)) \rightarrow \hat{a}(x,y-1,t-1) \\ (t = \lceil \frac{3z}{2} \rceil) \wedge (odd(z)) \rightarrow \hat{d}(x,y-1,t-2) \\ \lceil \frac{3z}{2} \rceil < t < 2z \rightarrow \hat{d}(x,y-1,t-2) \end{cases}$$

$$\hat{b}(x,y,t) = \begin{cases} (t = \lceil \frac{3z}{2} \rceil) \wedge (even(z)) \rightarrow \hat{e}(x-1,y,t-1) \\ (t = \lceil \frac{3z}{2} \rceil) \wedge (odd(z)) \rightarrow \hat{b}(x-1,y,t-2) \\ \lceil \frac{3z}{2} \rceil < t < 2z \rightarrow \hat{b}(x-1,y,t-2) \end{cases}$$

$$\hat{e}(x,y,t) = \begin{cases} t = 2z \rightarrow \hat{c}(x,y,t) \\ \lceil \frac{3z}{2} \rceil \leq t < 2z \rightarrow \hat{e}(x-1,y,t-1) \end{cases} \tag{6.4}$$

$$\hat{c}(x,y,t) = \begin{cases} z = 1 \rightarrow C_x \\ z > 1 \rightarrow \\ \quad \begin{cases} t = \lceil \frac{3z}{2} \rceil \rightarrow \\ \quad \min(h(\hat{a}(x,y,t), \hat{b}(x,y,t)), h(\hat{d}(x,y,t), \hat{e}(x,y,t)) \\ (\lceil \frac{3z}{2} \rceil \leq t < 2z) \rightarrow \\ \quad \min(\hat{c}(x,y,t-1), h(\hat{a}(x,y,t), \hat{b}(x,y,t)), \\ \quad h(\hat{d}(x,y,t), \hat{e}(x,y,t)) \\ t = 2z \rightarrow \hat{c}(x,y,t-1). \end{cases} \end{cases}$$

7. Target Systolic Architectures

7.1. Processor and time requirements

The design consists of a triangular array of $(n-1)(n-2)/2$ processors which complete the computation in $2(n-2)$ time steps, as indicated by the range of x, y and t in Proposition 6.1. Notice that the space-time mapping has achieved the goal of decreasing the complexity of processor requirement from $O(n^3)$ of the naive interpretation of System (5.5), to $O(n^2)$ in the new algorithm, by only increasing the time complexity with a constant factor, in this case, 2. Thus we see pipelining in the resulting algorithm and the successful sharing of a single processor among $O(n)$ processes.

7.2. I/O and storage requirements

In the resulting algorithm, if any of the space components of a process on the right-hand side of an equation differs from that on its left-hand side, then the data associated with the right-hand side process is

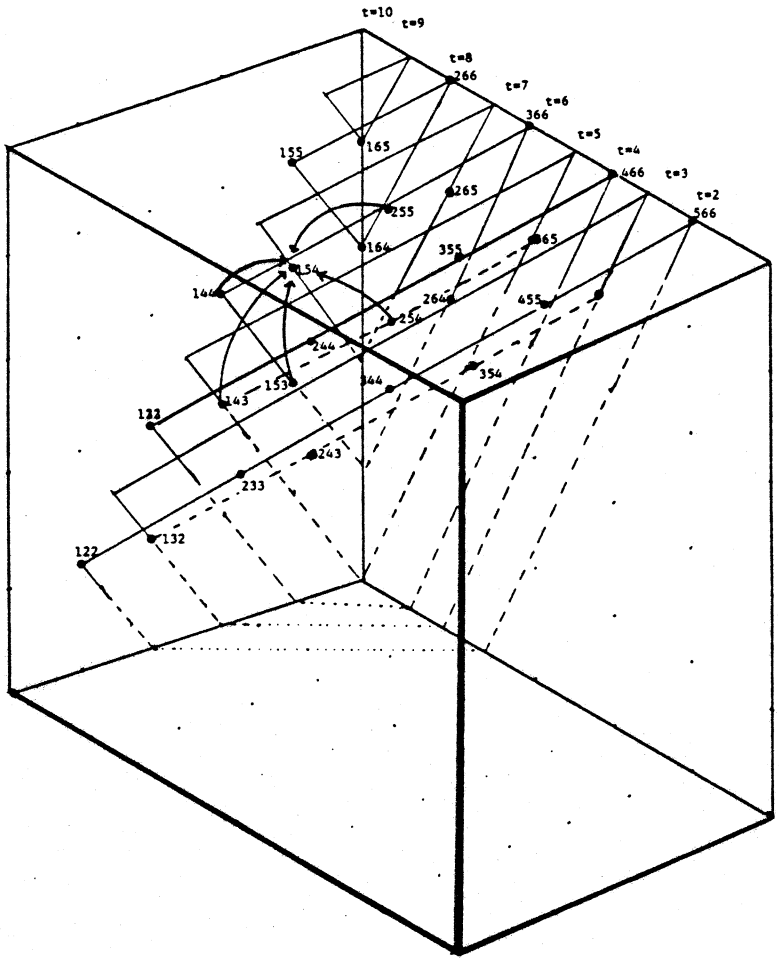

Figure 4: A series of parallel planes indicates the time steps of the space-time mapping. Processes on the same plane are mapped to the same invocation number t. (The processor number of a process in this case is its projection onto the bottom plane with the sign of x changed.)

an input to the left-hand side process, such as $\hat{a}(x, y-1, t-1)$ in Equation (6.4). In this case, the input is from processor $(x, y-1)$ which is, say, south of processor (x, y) and it takes one time step to arrive. If space components of two processes on both sides of an equation are the same, then the data associated with the right-hand side process is a stored value such as $\hat{c}(x, y, t-1)$ in Equation (6.4).

The space-time mapping in Proposition 6.1 yields the well-known systolic architecture for dynamic programming [7]. It can be seen in Figure 4 that a process (i, j, k) is mapped to processor $(-i, j)$ at time step $t = -2i + j + k$. Such an invocation may have inputs from invocations both at time $t-1$ and $t-2$ as shown, corresponding to the fast registers and the slow registers described in [7]. All signals controlling the loading and unloading between registers of different speeds can be systematically derived by techniques of program optimization as described below.

7.3. Control requirements

Predicates appearing in a given system of STREQ indicate the control of the functionality and timing of the processors. Their implementations might be different depending on different target implementation media. In a multiprocessor machine, the "processor id" (x, y) can be stored and an "invocation counter" keeps track of t until conditions such as $t = 2z$ are satisfied. In a VLSI implementation, however, it is too costly to store these integers and dedicate hardware to perform the tests. Program optimization of STREQ to eliminate such tests therefore becomes necessary. A better design can be obtained by replacing the expensive computations of a predicate by transferring a one-bit control signal, as illustrated by [7]. 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.

7.4. Control signal optimization

Definition 7.1. A *control expression* is an expression $p(x_1, x_2, \ldots, x_n)$ of the space indices x_i, $1 \leq i \leq n$, which is non-negative, rational-valued, and consists of only rational coefficients and first-power terms. However, the expression may consist of non-linear operations, e.g., absalute value, ceiling, floor, conditional, etc.

Proposition 7.1. *Suppose a control expression $p(x_1, x_2, \ldots, x_n)$ is strictly monotonic in some space index x_i, i.e., if $x_i' < x_i$ then $p(x_1, \ldots, x_i', \ldots, x_n) < p(x_1, \ldots, x_i, \ldots, x_n)$. Then for each x_i, the time dependency*

$$dt(x_i) = p(x_1, x_2, \ldots, x_i, \ldots, x_n) - p(x_1, x_2, \ldots, x_i - 1, \ldots, x_n)$$

of the control expression has a fixed value which is positive.

Proposition 7.2. *Suppose a control expression $p(x_1, x_2, \ldots, x_n)$ is strictly monotonic descreasing in some space index x_i, i.e., if $x_i' < x_i$ then $p(x_1, \ldots, x_i', \ldots, x_n) > p(x_1, \ldots, x_i, \ldots, x_n)$. Then for each x_i, the time dependency*

$$dt(x_i) = p(x_1, x_2, \ldots, x_i, \ldots, x_n) - p(x_1, x_2, \ldots, x_i + 1, \ldots, x_n)$$

of the control expression has a fixed value which is positive.

Theorem 7.1. *Time-variant predicates of the form*

$$t < p(x_1, x_2, \ldots, x_n),$$
$$t = p(x_1, x_2, \ldots, x_n), \text{ and,}$$
$$t > p(x_1, x_2, \ldots, x_n),$$

where $p(x_1, x_2, \ldots, x_n)$ is a strictly monotonic control expression, can be implemented by time-invariant

predicates

$$q(x_1, x_2, \ldots, x_n, t) = 0,$$

$$q(x_1, x_2, \ldots, x_n, t) = 1, \text{ and}$$

$$q(x_1, x_2, \ldots, x_n, t) = 2, \text{ respectively,}$$

where $q(x_1, x_2, \ldots, x_n, t)$ *is a control stream*

$$q(x_1, x_2, \ldots, x_n, t) = \begin{cases} p(x_1, x_2, \ldots, x_n) = 0 \rightarrow \begin{cases} t < 0 \rightarrow 0 \\ t = 0 \rightarrow 1 \\ t > 0 \rightarrow 2 \end{cases} \\ \quad \textit{(establish the truth initially for the test)} \\ p(x_1, x_2, \ldots, x_n) > 0 \rightarrow q(x_1, x_2, \ldots, x_i - 1, \ldots, x_n, t - dt(x_i)) \end{cases}$$

in which all predicates, except for the switch-on predicates $t < 0$, $t = 0$, *and* $t > 0$, *are time-invariant, and* $dt(x_i)$ *is the time dependency of control expression* $p(x_1, x_2, \ldots, x_n)$.

Certainly, a similar theorem holds for a control expression that is strictly monotonic decreasing. For an application of the above theorem, take for instance, predicates $t = x - 1$ and $t > x - 1$. The control expression $x - 1$ is monotonic in x, and its time dependency is $dt(x) = 1$ for all x. By the theorem, it can be implemented by predicates $q(x, t) = 1$ and $q(x, t) = 2$, respectively, where the control stream is defined as

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

Note that since predicate $t < x - 1$ is not needed, $q(x, t)$ needs to assume only two values, 1 and 2. Thus it can be reduced to a single bit signal. This control stream can be easily implemented by a one-bit signal initially fed to processor $x = 1$, which changes its value at $t = 1$, 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 $q(x_1, x_2, \ldots, x_n, t)$ 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.

In general, a control expression might be strictly monotonic (decreasing) in more than one space index. The choice of an index x_i should be such that $dt(x_i)$ is minimized over all other choices of space indices because $dt(x_i)$ can determine the rate at which data streams are transferred. In the case when some choices of space indices are equally good as far as timing is concerned, the topology of the chip layout and the input pin arrangement may determine which space index should be chosen since control signals must be fed into the chip from outside.

For the dynamic programming example, one possible implementation of all of the time-variant predicates is illustrated in BOX 5 of the appendix. It is worth noting, in particular, the implementation of the predicate

$t = \lceil \frac{3z}{2} \rceil$. The predicate contains a piece-wise linear control expression, and the resulting signal travels in those processors with odd z in half the speed as it does in those processors with even z. This particular type of signal is described incorrectly as a linear signal that moves "two cells every three time units" in [7] as opposed to the above-mentioned piece-wise linear signal.

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.

7.5. Transformations for lower dimensional networks

Another stage of transformations is required if a problem must be solved on a fixed-dimensional network. Taking the dynamic programming problem for instance, suppose one is intereseted in solving it by a one-dimensional network. At any given time step of the target program above, there are $O(n^2)$ number of processors in execution, and the number must be reduced to $O(n)$ for a one-dimensional network. For any processor (x, y) in this target program, one can simply choose to interpret any of the space components, say, x as a loop index within a processor y (described by the rest of space components). It is clear then that any linear combination of the space components can serve as a loop index. In general, one can perform a linear transformation on the original space coordinates and choose any one of the components of the new coordinates as a loop index; the remaining components then become the new "processor id". The transformations on the program are quite similar to those described in Section 6.6.

8. Related Work

Many attempts in pursuit of systematic methods for synthesizing systolic computations have appeared in the literature. In the classification of systolic designs from the geometric point of view, Lin and Mead [10] classify systolic arrays using symmetry groups, but there is no attempt made to obtain new designs. Cappello and Steiglitz [1] view systolic designs as affine transforms from combinational designs; however, their method does not give clues to finding the appropriate transforms.

Moldovan [12, 13] notes that data dependency vectors are what suggest clues to potential linear transforms. Miranker and Winkler [11] describe a similar method, and they have observed that all planar two-dimensional regularly connected arrays can be embedded in a hexagonal array, called a universal array. Quinton [14] starts with a specification called a system of uniform recurrence equations and applies affine and linear transforms to obtain a systolic array. The class of systolic computations that can be described by uniform recurrence equations is limited to those that have only one of their data streams dependent upon other data streams. Li and Wah [8] describe a method that first finds an appropriate placement of one of the input streams and then perform the linear transforms. Delosme and Ipsen [6] describe how to implement a hyperbolic Cholesky solver by a linear transformation from a system of recurrences.

All of the above four methods focus on (quasi-)linear or affine mappings from a restricted class of

algorithms to a new design. The class of designs that can be obtained by these methods are subject to restrictions such as starting with uniform recurrent equations (a much more restrictive definition than the uniformity defined in this paper), yielding only designs that are synchronous, regular, having uniform data flow, etc. The capabilities of these methods in various restricted situations may, at best, be equal to the general mapping procedure for the uniform algorithms defined in this paper. However, instead of using a simple, constant time procedure consisting of enumerating the basis communication vectors by subgroup enumerations and finding mappings between the two sets of basis vectors, they find linear mappings by searching through the space of possible linear transforms or input placements. The search requires a time that is at least polynomial of the size of the problem [8].

Some attempts, such as in [12, 11] are made to transform an initial mathematical definition to a description that is suitable for space-time mapping. The method used in their work applies to certain specific cases, and is somewhat *ad hoc* and not systematic or foraml enough for potentially automating the process of transformations.

Li and Wah [9] have classified the problems that can be solved by dynamic programming into several classes and have proposed systolic processors for solving them. The formulation given in this paper corresponds to the most general class — in their terminology, the "polyadic-nonserial" class. In their treatment, the issue of synchronization and timing is not addressed.

9. Concluding Remarks

The objective of this work is the devising of efficient parallel programs and VLSI architectures. Due to the complexity that might arise in dealing with hundreds of thousands of autonomous parallel processing elements, we find design methodology for this process necessary in order to take the enormous burden off the designers.

Program synthesis, at the very basic level, relies on a formal notation for describing the problem, and henceforth on manipulating the descriptions to yield efficient parallel programs. This paper describes, along with a set of program synthesis methods, a design methodology and program transformation environment in which these methods are applied at the various stages of transformations, such as the stage that proceeds from a problem definition to a program that has bounded fan-in and fan-out degrees, the stage that reduces long-range communications to local ones, the stage that incorporates pipelining into programs by space-time mapping, and the stage that derives optimized control signals.

It is worth mentioning that the transformation rules to derive bounded fan-in/fan-out degree and bounded order recursion equations are automatable and apply to problem definitions in general. The procedure of space-time mapping for uniform algorithms and the test for the existence of linear mappings are computationally attractive and their complexities are independent of the problem sizes. The theorems that allow the conversion of time-variant predicates to time-invariant ones for generating optimized control signals have a wide range of applications.

It is essential that the language **Crystal** be amenable to algebraic manipulation so that techniques developed for algorithm transformation can be carried out formally by an automated process. More importantly, **Crystal** 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.

10. Appendix

The target program enclosed in the box shall be replaced subsequently by code in BOX 1 to BOX 5, one by one. Each replacement results in a new **Crystal** program that is more optimized towards the final implementation. The result of each program is requested in such a way that all programs give the same outputs, as shown at the bottom of Figure 5.

Figure 5: Dynamic programming in **Crystal**.

References

[1] Cappello, P.R. and Steiglitz, K., *Unifying VLSI Array Designs with Geometric Transformations,* IEEE Proceedings of the International Conference on Parallel Processing, (1983).

[2] 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.

[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] ————, Transformations of Parallel Programs in Crystal, *The Proceedings of the IFIP 86, Dublin, Ireland,* September 1986.

[6] 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.

[7] Guibas, L. J. Kung, H. T. and Thompson, C. D., Direct VLSI Implementation of Combinatorial Algorithms, *Proc. Caltech Conf. VLSI,* 1979.

[8] 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.

[9] ————, Systolic Processing for Dynamic Programming Problems, *Proceedings of the 1985 International Conference on Parallel Processing,* 1985, pp. 434–441.

[10] Lin, T.Z. and Mead, C.A., *The Application of Group Theory in Classifying Systolic Arrays,* Display File 5006, Caltech, Mar 1982.

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

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

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

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

```
! DYNAMIC PROGRAMMING

! output the set of costs on all pairs (i,j)

{ [i,j,C(i,j)] | (i in 1 : n-1, j in 2 : n), i < j }
where (

! problem size
   n = 6

  MAXINT = 99999

! top level program
! target of program transformation starts here
! note that the identifiers used in the programs may not be the same as
! those presented in the paper due to the restriction imposed by the
! character set.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                                                                          !
! this is a general definition for a class of dynamic programming problems !
                                                                          !
! minimizing cost C(i,j)                                                  !
                                                                          !
    C(i,j) =                                                              !
      << (j-i)-1 = 0 -> C0 (i),                                           !
         (j-i)-1 > 0 -> \min_c { [h(C(i,k), C(k,j)), k] | k in (i+1):(j-1) }  !
      >>                                                                  !
                                                                          !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! target of program transformation ends here
! the following are definitions used by the top level program

    where (

! EXAMPLE: optimal parenthesization for a sequence of matrix multiplications.
    ! The format of a cost C(i,j) is [[ [p, r], c], k] where
    ! [p, r] is the pair of dimensions of the resulting matrix product
    ! of matrix i through matrix j-1.
    ! k records the optimal location for subdividing the sequence into
    ! two subsequences.
    ! c stores the optimal accumulated cost for multiplying
    ! matrix i through matrix j-1 via the optimal subdivisions at k.


    ! INITIAL COST  C0(i).
    C0(i) = [[ref(dim, i), 0], 0]
    where dim = [ [15, 4], [4, 8], [8, 13], [13, 9], [9, 6] ]
    ! ref is a function which accesses a component of a vector.
    ! dim is a vector which contains the pairs of dimensions of the
    ! input matrices to be multiplied.

    ! BINARY OPERATOR ON COSTS
    min_c(a, b) =
       << (a = NIL) or (b = NIL) -> id_min,
          else ->
           << ref(a, [1,2]) <= ref(b, [1,2]) ->  a,
              else -> b
           >>
       >>
    where id_min = [[[MAXINT, MAXINT], 0], 0]
    ! the identity element for the binary operator min_c


    ! FUNCTION FOR COMBINING COSTS u and v from sub_problems:
    ! assume the cost of multiplying a (p x q) matrix by a (q x r) matrix
    ! is p*q*r, c1 and c2 are costs accumulated from previous stages, and
    ! [p, r] is the pair of dimensions of the new matrix.

    h(u, v) = [ [p, r],  c1 + c2 + p * q * r]
       where (
             p = ref(u, [1,1,1])
             r = ref(v, [1,1,2])
             q = << ref(u, [1,1,2]) = ref(v, [1,1,1]) -> ref(v, [1,1,1]),
                    else -> error ! error in inputs
                                    where error = 99999
                 >>
             c1 = ref(u, [1,2])
             c2 = ref(v, [1,2])
           ) ! end of where for h

         ) ! end of where for C

  )

! Result from executing the program

RESULT:
{[1, 2, [[[15, 4], 0], 0]]
 [1, 3, [[[15, 8], 480], 2]]
 [1, 4, [[[15, 13], 1196], 2]]
 [1, 5, [[[15, 9], 1424], 2]]
 [1, 6, [[[15, 6], 1460], 2]]
 [2, 3, [[[4, 8], 0], 0]]
 [2, 4, [[[4, 13], 416], 3]]
 [2, 5, [[[4, 9], 884], 4]]
 [2, 6, [[[4, 6], 1100], 5]]
 [3, 4, [[[8, 13], 0], 0]]
 [3, 5, [[[8, 9], 936], 4]]
 [3, 6, [[[8, 6], 1326], 4]]
 [4, 5, [[[13, 9], 0], 0]]
 [4, 6, [[[13, 6], 702], 5]]
 [5, 6, [[[9, 6], 0], 0]]}
**VALUE-OF-FORMAT**
```

Figure 5: A program of dynamic programming in Crystal.

```
!BOX 1

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This systems of recurion equations has constant fan-out degree

   a(i,j,k) = << j = k -> C(i,k),
                 j > k -> a(i, j-1, k)
              >>

   b(i,j,k) = << i = k -> C(k,j),
                 i < k -> b(i+1, j, k)
              >>

   C(i,j) =
     << (j-i)-1 = 0 -> C0 (i),
        (j-i)-1 > 0 ->
          \min_c { [h(a(i,j,k), b(i,j,k)), k] | k in (i+1):(j-1) }
     >>

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!BOX 2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This systems of recurion equations has constant fan-in/fan-out degrees

   C(i,j) = cc(i,j,j)

   ! substituting cc into a and b
   a(i,j,k) = << j = k -> cc(i,k,k),
                 j > k -> a(i, j-1, k)
              >>

   b(i,j,k) = << i = k -> cc(k,j,j),
                 i < k -> b(i+1, j, k)
              >>

   ! fan-in degree as been reduced
   cc(i,j,k) =
     << (j-i)-1 = 0 -> C0 (i),

        (j-i)-1 > 0 ->
         << k = m(i,j) -> min_c( [h(a(i,j,k), b(i,j,k)), k],
                        [h(a(i,j, (i+j)-k), b(i,j, (i+j)-k)), (i+j)-k] ),

            m(i,j) < k < j ->
              \min_c { cc(i,j,k-1), [h(a(i,j,k), b(i,j,k)), k],
                        [h(a(i,j, (i+j)-k), b(i,j, (i+j)-k)), (i+j)-k] },

            k = j -> cc(i,j, k-1)
         >>
        where ( m(i,j) = << even(i+j) -> (i+j)/2,
                            odd(i+j) -> (i+j+1)/2
                         >>
              )
     >>

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!BOX 3

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This systems of recurion equations is of bounded order

   C(i,j) = cc(i,j,j)

! domain contraction
! transform a into a and d, b into b and e

   ! auxiliary function m
   m(i,j) = << even(i+j) -> (i+j)/2,
               odd(i+j) -> (i+j+1)/2
                     >>

   a(i,j,k) = << k = j -> cc(i,k,k),
                 m(i,j) <= k < j -> a(i, j-1, k)
              >>

   d(i,j,k) = << (k = m(i,j)) and (odd((j-i)-1)) -> a(i, j-1, k),
                  (k = m(i,j)) and (even((j-i)-1)) -> d(i, j-1, k-1),
                 m(i,j) < k < j -> d(i, j-1, k-1)
              >>

   b(i,j,k) = << (k = m(i,j)) and (odd((j-i)-1)) -> e(i+1, j, k+1),
                  (k = m(i,j)) and (even((j-i)-1)) -> b(i+1, j, k),
                 m(i,j) < k < j -> b(i+1, j, k)
              >>

   e(i,j,k) = << k = j -> cc(i,j,j),
                 m(i,j) <= k <j -> e(i+1, j, k+1)
              >>

   ! substituting new d and e into cc
   cc(i,j,k) =
     << (j-i)-1 = 0 -> C0 (i),

        (j-i)-1 > 0 ->
         << k = m(i,j) -> min_c( [h(a(i,j,k), b(i,j,k)), k],
                                 [h(d(i,j,k), e(i,j,k)), (i+j)-k] ),

            m(i,j) < k < j ->
              \min_c { cc(i,j,k-1), [h(a(i,j,k), b(i,j,k)), k],
                                    [h(d(i,j,k), e(i,j,k)), (i+j)-k] },

            k = j -> cc(i,j, k-1)
         >>
     >>

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!BOX 4

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Space-time recursion equations (STREQ)                               !
                                                                       !
   C(i,j) = cc( g_x(i, j, j), g_y(i, j ,j), g_t(i,j,j) )               !
                                                                       !
   ! space-time mapping                                                !
   g_x(i,j,k)  = -i   g_y(i,j,k) = j    g_t(i,j,k) = (-2 * i) + j + k  !
                                                                       !
   ! inverse mapping                                                   !
   gv_i (x,y,t) = -x  gv_j (x,y,t) = y  gv_k (x,y,t) = t - (2 * x + y) !
                                                                       !
                                                                       !
   !  auxiliary functions                                              !
   z(x,y) = x + y                                                      !
                                                                       !
   m(x,y) = << even(z(x,y)) -> (3*z(x,y))/2,                           !
               odd(z(x,y)) -> ((3*z(x,y))+1)/2                         !
                               >>                                      !
   ! Space-time recursion equations (STREQ)                            !
                                                                       !
   a(x,y,t) = << t = 2*z(x,y) -> cc(x,y,t),                            !
                  m(x,y) <= t < 2*z(x,y) -> a(x, y-1, t-1)             !
               >>                                                      !
                                                                       !
   d(x,y,t) = << (t = m(x,y)) and (even(z(x,y))) -> a(x, y-1, t-1),    !
                 (t = m(x,y)) and (odd(z(x,y))) -> d(x, y-1, t-2),     !
                 m(x,y) < t < 2*z(x,y) -> d(x, y-1, t-2)               !
               >>                                                      !
                                                                       !
   b(x,y,t) = << (t = m(x,y)) and (even(z(x,y))) -> e(x-1, y, t-1),    !
                 (t = m(x,y)) and (odd(z(x,y))) -> b(x-1, y, t-2),     !
                 m(x,y) < t < 2*z(x,y) -> b(x-1, y, t-2)               !
               >>                                                      !
                                                                       !
   e(x,y,t) = << t = 2*z(x,y) -> cc(x,y,t),                            !
                 m(x,y) <= t < 2*z(x,y)-> e(x-1, y, t-1)               !
               >>                                                      !
                                                                       !
   cc(x,y,t) =                                                         !
     << z(x,y) = 1 -> C0(gv_i(x,y,t)),                                 !
                                                                       !
        z(x,y) > 1 ->                                                  !
         << t = m(x,y) -> min_c( [h(a(x,y,t), b(x,y,t)), gv_k(x,y,t)], !
                                 [h(d(x,y,t), e(x,y,t)),               !
                                 gv_i(x,y,t)+gv_j(x,y,t)- gv_k(x,y,t)] ),!
                                                                       !
            m(x,y) < t < 2*z(x,y) ->                                   !
              \min_c { cc(x,y,t-1), [h(a(x,y,t), b(x,y,t)), gv_k(x,y,t)],!
                                    [h(d(x,y,t), e(x,y,t)),            !
                                 gv_i(x,y,t)+gv_j(x,y,t)- gv_k(x,y,t)] },!
                                                                       !
            t = 2*z(x,y) -> cc(x, y, t-1)                              !
          >>                                                           !
      >>                                                               !
                                                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!BOX 5

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Space-time recursion equations without time-variant predicates      !
                                                                       !
   C(i,j) = cc( g_x(i, j, j), g_y(i, j ,j), g_t(i,j,j) )               !
                                                                       !
   ! space-time mapping                                                !
   g_x(i,j,k)  = -i   g_y(i,j,k) = j    g_t(i,j,k) = (-2 * i) + j + k  !
                                                                       !
   ! inverse mapping                                                   !
   gv_i (x,y,t) = -x  gv_j (x,y,t) = y  gv_k (x,y,t) = t - (2 * x + y) !
                                                                       !
                                                                       !
   !  auxiliary functions                                              !
   z(x,y) = x + y                                                      !
                                                                       !
   m(x,y) = << even(z(x,y)) -> (3*z(x,y))/2,                           !
               odd(z(x,y)) -> ((3*z(x,y))+1)/2                         !
                               >>                                      !
                                                                       !
   ! Now replacing time-variant predicates with control streams        !
   ! which themselves do not have any time-variant predicate           !
                                                                       !
   ! New auxiliary functions                                           !
   dt(x,y) = << even(z(x,y)) -> 1,                                     !
                else -> 2         >>  ! this is m(x,y) - m(x-1, y)     !
   dt2 = 2       ! this is  2 * z(x,y) - 2 * (x-1, y)                  !
                                                                       !
   ! New control stream for implementing t = m(x,y)                    !
   u(x,y,t) = << m(x,y) = 0 -> << t = 0 -> 1,                          !
                                  t > 0 -> 0                           !
                               >>,                                     !
                 m(x,y) > 0 -> u(x-1, y, t-dt(x,y) )                   !
               >>                                                      !
                                                                       !
   ! New control stream for implementing t = 2 * z(x,y)                !
   v(x,y,t) = << 2*z(x,y) = 0 -> << t = 0 -> 1,                        !
                                    t < 0 -> 0    ! by initialization  !
                                 >>,                                   !
                 2*z(x,y) > 0 -> v(x-1, y, t- dt2)                     !
               >>                                                      !
                                                                       !
  ! Space-time recursion equations without time-variant predicates     !
                                                                       !
   a(x,y,t) = << v(x,y,t) = 1 -> cc(x,y,t),                            !
                 (u(x,y,t) <= 1) and  (v(x,y,t) = 0) -> a(x, y-1, t-1) !
               >>                                                      !
                                                                       !
   d(x,y,t) = << (u(x,y,t)=1) and (even(z(x,y))) -> a(x, y-1, t-1),    !
                 (u(x,y,t)=1) and (odd(z(x,y))) -> d(x, y-1, t-2),     !
                 (u(x,y,t)=0) and (v(x,y,t)=0) -> d(x, y-1, t-2)       !
               >>                                                      !
                                                                       !
   b(x,y,t) = << (u(x,y,t)=1) and (even(z(x,y))) -> e(x-1, y, t-1),    !
                 (u(x,y,t)=1) and (odd(z(x,y))) -> b(x-1, y, t-2),     !
                 (u(x,y,t)=0) and (v(x,y,t)=0) -> b(x-1, y, t-2)       !
               >>                                                      !
```

```
e(x,y,t) = << v(x,y,t) = 1 -> cc(x,y,t),
              (u(x,y,t) <= 1) and (v(x,y,t)=0)-> e(x-1, y, t-1)
           >>

cc(x,y,t) =
  << z(x,y) = 1 -> C0(gv_i(x,y,t)),

     z(x,y) > 1 ->
      << u(x,y,t) = 1 -> min_c( [h(a(x,y,t), b(x,y,t)), gv_k(x,y,t)],
                                [h(d(x,y,t), e(x,y,t)),
                                gv_i(x,y,t)+gv_j(x,y,t)- gv_k(x,y,t)] ),

         (u(x,y,t)=0) and (v(x,y,t)=0)->
           \min_c { cc(x,y,t-1), [h(a(x,y,t), b(x,y,t)), gv_k(x,y,t)],
                                 [h(d(x,y,t), e(x,y,t)),
                                gv_i(x,y,t)+gv_j(x,y,t)- gv_k(x,y,t)] },

         v(x,y,t) = 1 -> cc(x, y, t-1)
       >>
  >>
```