Yale University
Department of Computer Science

A Synthesis Approach to the Design and Correctness of Systolic Computations

Marina C. Chen

YALEU/DCS/TR-477
June 1986

Work supported by NSF Grant DCR 8106181 and ONR Contract N0014-85-K-0030.

A Synthesis Approach to the Design and Correctness of Systolic Computations

Marina C. Chen[1]

Abstract

This paper presents a methodology for deriving systolic algorithms by a series of transformations from mathematical problem definitions. A conceptualization of systolic computations, in terms of mathematical objects rather than merely in terms of implementations, is provided. The concept of wavefront sequence is introduced to construct timing functions for synchronizing parallel processes. Its use, however, is not limited to systolic computations but also serves as a reasoning tool for parallel computations in general. A complete example, from the definition of matrix multiplication by inner products to the derivation of its four different systolic implementations, is given. The correctness of the original definition is strictly maintained at each stage of the transformations.

1 Introduction

Parallel processing is concerned with the effective use of hardware technologies that are inherently highly parallel. In order to expoit its potential fully, parallelism must be used at the algorithmic level as well as at the hardware level. As a consequence, parallel software technologies that facilitate the use of parallel hardware play an increasingly more important role in the arena of parallel processing.

Systolic algorithms, as exemplified by a large body of literature [9], make use of hardware technology very effectively in several aspects: (1) they avoid the incurrence of high communication cost 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).

Can such efficiency be incorporated into a program with relative ease or must each systolic algorithm be obtained painstakingly in an ad hoc fashion? Can the correctness of a systolic algorithm be ensured in a formal and rigorous manner instead of simulation? Unlike a sequential program where the computation it embodies can be modeled as a sequence of state changes and thus be reasoned with existing programming methodologies, the concurrent and distributed state changes appearing in a parallel program make the reasoning process a much more difficult one. A great deal of work in the areas of proof methods, logic of concurrent programs, and verification techniques for concurrent or parallel programs has appeared [1], [4], [7], [10], [11], [12], [13]. These methods provide foundations for proving the correctness of concurrent programs, but they aim at showing the correctness upon being given a concurrent program rather than allowing an efficient parallel program to be synthesized.

In this paper, we present a synthesis approach for *deriving* systolic algorithms by a series of transformations from a problem definition. We illustrate how to extract a "sequence of execution wavefronts" from a parallel program and use it as a tool for reasoning about the

[1]Department of Computer Science, Yale University, New Haven, CT 06520. Work supported in part by the National Science Foundation under Grant No. MCS-8106181 (DCR-8106181) and the Office of Naval Research under Contract No. N00014-85-K-0030.

program. A problem definition is given in the language **Crystal**, which is essentially a formalized mathematical notation. The correctness of a definition is either self-evident, or it is a scientifically well-established fact, or else it can be proven correct mathematically with relative ease. In each step of the transformation, a rule aiming at reducing the underlying hardware cost in the parallel implementation is introduced, and the correctness of the definition is strictly maintained.

The remaining parts of this paper are organized as follows: In Section 2, a special class of **Crystal** programs that yield systolic computations is defined. In Section 3, examples of problem definitions are given. Each problem definition, upon being broken down into various parts of a **Crystal** program, immediately exhibits the large scale parallelism it contains. In Section 4, notions relating to the ordering of a large scale of processes such as process structures and wavefront sequences are introduced. In Section 5, the issue of effective utilization of hardware technologies is addressed. Synthesis rules for transforming programs to ones suitable for implementation are introduced. In Section 6, the synchronization of a large number of processes, the opitmal timing function, and the space-time mapping of a program are introduced. The existence and uniqueness of the optimal timing funtion of a program are established. By space-time mapping, automatic incorporation of pipelining into programs is achieved. A resulting program has improved efficiency and embodies a systolic computation.

Throughout this paper, the example of deriving several systolic algorithms from the mathematical definition of matrix multiplication is used. First, in Section 5.7, a bounded-order and bounded fan-in and fan-out degree program $P_{mm'}$ is derived from the initial matrix multiplication definition P_{mm}. Next, in Section 6.3, the optimal timing function for $P_{mm'}$ is constructed. Several optimal space-time mappings are then constructed from the optimal timing function. Finally, in Section 7, initialization, inputs, and outputs that suit the implementations of a two-dimensional network are incorporated into each target program.

2 A Special Class of Crystal Programs

For the scope of this paper, a special class of **Crystal** programs, the ones with *data-independent process structures* [2] is defined. A general definition of a **Crystal** program can be found in [6]. A systolic computation, or any fine-grain parallel computation, relies on the parallelism operated on a large quantity of data. At the very basic level of any formalism for describing such class of computations, it is important that each datum can be addressed in a structured and convenient way so that parallelism over the collection of datum can be expressed appropriately. This property of fine-grain parallelism motivates the following definition:

Definition 2.1 (Index Set) Let A_i be a Cartesian product $A_{i1} \times A_{i2} \times \cdots \times A_{iq}$ of countable sets A_{ij}, $1 \leq j \leq q$ and q a positive integer, which are usually subsets of the set of integers. The set $(A_{ij}, \sqsubseteq)$ with the binary relation "approximate" ([14]) on the elements of A_{ij} is a flat lattice. An *index set* A is a sum $A_1 + A_2 + \cdots + A_r$ of Cartesian products $(A_i, \sqsubseteq)$, for some positive integer r, and any subset of an index set is also an index set.

A **Crystal** program can be quite a complex object since a parallel program may consist of a large number of processes each of which is performing a different task and communicating with others via various communication paths. Informally, a **Crystal** program can be depicted as shown in Figure 1, where $\mathbf{v}_j$ are processes, F_i are data streams, X_i are external input functions, $\mathbf{m}_k$ are elements of the output data structure, and the output mapping function o maps a given $\mathbf{m}_k$ to a particular process $o(\mathbf{m}_k)$ at which the value $F_i(o(\mathbf{m}_k))$ are of interest and considered external outputs of the program. Note that the inputs of a process $\mathbf{v}_j$ may be the outputs $F_i(\mathbf{u})$

[2] Definition in Section 4.

of some other process $\mathbf{u}$, or may be some external input value $X_i(\mathbf{b})$, where $\mathbf{b}$ is an element of the domain of X_i. Note that here we are taking a functional programming point of view in which each process performs some local processing function ($\phi_{\mathbf{v}}$ defined below), and the data flow is specified by the arrows connecting processes (communication functions $\tau_{\mathbf{v}}$ defined below) . A process may start processing as long as all its required inputs are available and no particular timing constraints are imposed on the collection of processes. Issues of optimal synchronization or timing are dealt with when processes are mapped to processors in Section 6.3.

Formally, a **Crystal** program has two parts, the declaration and the program body. Program declaration consists of a 9-tuple,

$$\mathcal{P} = (P, \mathcal{D}, \mathcal{F}, \mathcal{X}, M, \{\phi_{\mathbf{v}}\}_{\mathbf{v}\in P}, \{\tau_{\mathbf{v}}\}_{\mathbf{v}\in P}, \iota, o),$$

and the program body is a system of recursion equations

$$\mathbf{F}(\mathbf{v}) = \phi_{\mathbf{v}}(\mathbf{F}\langle\tau_{\mathbf{v}}(\mathbf{v})\rangle, \mathbf{X}\langle\iota(\mathbf{v})\rangle), \ \forall \mathbf{v} \in P, \tag{1}$$

where

1. P is an index set, and each element $\mathbf{v} \in P$ is called a process. The set of processes P with some ordering of these processes (to be defined later) is referred to as the *process structure* of program $\mathcal{P}$.

2. $\mathcal{D}$ is a set of domains (data types). Each domain $D_i \in \mathcal{D}$ can be some value domains such as the set of integers, reals, etc., or it can be some domain of functions, functions of functions, etc., and $(D_i, \sqsubseteq)$ is a continuous complete lattice.

3. $\mathcal{F}$ is a set of identifiers for *data streams*, where a data stream is an element of the set $E_i \stackrel{\text{def}}{=} [(P, \sqsubseteq) \to (D_i, \sqsubseteq)]$ of continuous functions from $(P, \sqsubseteq)$ to $(D_i, \sqsubseteq)$.

4. $\mathcal{X}$ is a set of identifiers for *input functions*. Each input function $X_i \in \mathcal{X}$ is an element of the set of functions $[(B, \sqsubseteq) \to (D, \sqsubseteq)]$, where D is some data type in $\mathcal{D}$ and B is an index set, called the *input data structure* of program $\mathcal{P}$.

5. M is the output data structure, which is an index set.

6. In the following definitions, m and k are some positive integers and l is a non-negative integer. These are the three natural numbers that specify the number of data streams, the number of external outputs, and the number of external input functions of a given **Crystal** program.

7. If f is a function of variable x defined by an expression E, then we use the notation $\lambda x.E$ to denote f, i.e., $f = \lambda x.E$, and the functional value $f(x)$ is denoted by $(\lambda x.E)(x)$.

8. A *local processing function* $\phi_{\mathbf{v}} = [\phi_{\mathbf{v},1}, \ldots, \phi_{\mathbf{v},m}]$ is a member of the family of functions $\{\phi_{\mathbf{v}}\}_{\mathbf{v}\in P}$, and each of its component $\phi_{\mathbf{v},i}$, for $1 \le i \le m$, is a function

   $$\phi_{\mathbf{v},i} : D_1 \times \cdots \times D_m \times D'_1 \times \cdots \times D'_l \to D_i,$$

   and

   $$\phi_{\mathbf{v},i}(d_1, \ldots, d_m, d'_1, \ldots, d'_l) = e'_i, \text{ where } d_i, e_i \in D_i, \text{ and } d'_i \in D'_i.$$

9. A *communication function* $\tau_{\mathbf{v}} = [\tau_{\mathbf{v},1}, \ldots, \tau_{\mathbf{v},m}]$ is a member of a family of functions $\{\tau_{\mathbf{v}}\}_{\mathbf{v}\in P}$, and each of its components $\tau_{\mathbf{v},i}$, for $1 \le i \le m$ is a function $\tau_{\mathbf{v},i} : P \to P$ of the form $\lambda \mathbf{u}.E(\mathbf{u})$. This definition of communication function specializes a **Crystal** program to the class of programs with data-independent process structures due to its independence of any value of any data stream $F(\mathbf{v})$ since $\tau_{\mathbf{v},i}$ is a function only of $\mathbf{v} \in P$.

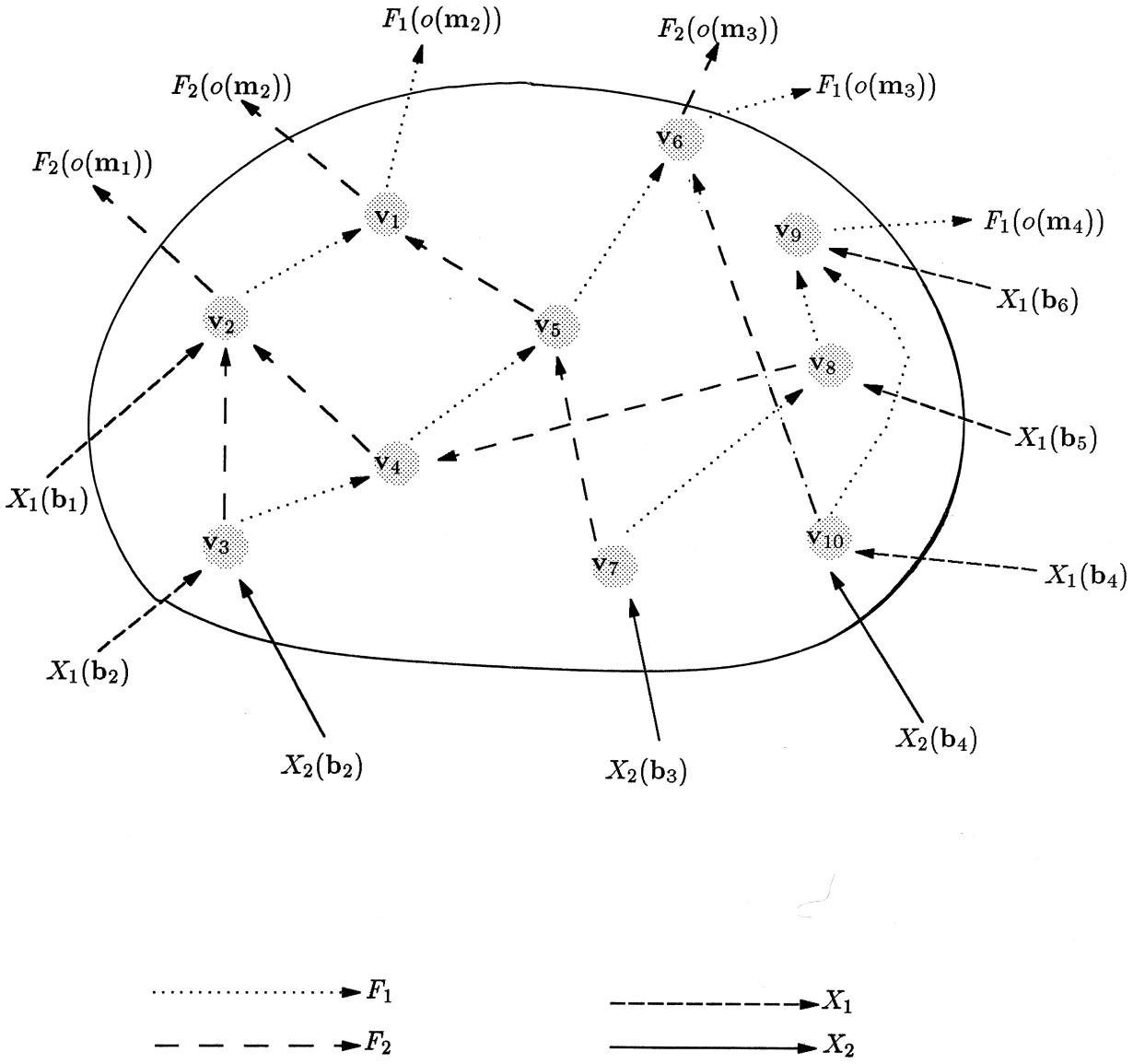

Figure 1: A **Crystal** program consisting of processes, communications between processes, data streams, external inputs, input mapping function ι which maps $\mathbf{v}_2 \mapsto \mathbf{b}_1$, $\mathbf{v}_3 \mapsto \mathbf{b}_2$, $\mathbf{v}_7 \mapsto \mathbf{b}_3$, $\mathbf{v}_{10} \mapsto \mathbf{b}_4$, $\mathbf{v}_8 \mapsto \mathbf{b}_5$, $\mathbf{v}_9 \mapsto \mathbf{b}_6$, external outputs, and output mapping function o which maps $\mathbf{m}_1 \mapsto \mathbf{v}_2$, $\mathbf{m}_2 \mapsto \mathbf{v}_1$, $\mathbf{m}_3 \mapsto \mathbf{v}_6$, $\mathbf{m}_4 \mapsto \mathbf{v}_9$.

10. $\mathbf{X} \stackrel{\text{def}}{=} [X_1, \ldots, X_l]$ is a vector of input functions where $X_i \in [B \rightarrow D_i']$, $1 \leq i \leq l$.

11. $\mathbf{F} \stackrel{\text{def}}{=} [F_1, \ldots, F_m]$ is a vector of data streams where $F_i \in E_i$ (defined in Item 3), $1 \leq i \leq m$.

12. $\mathbf{F}(\mathbf{v}) \stackrel{\text{def}}{=} [F_1(\mathbf{v}), \ldots, F_m(\mathbf{v})]$.

13. Function $\iota = [\iota_1, \ldots, \iota_l]$ is a vector of *input mapping functions* where each of its components $\iota_j : P \rightarrow B$, $1 \leq j \leq l$, is an input mapping function which interfaces the input data structure and the process structure.

14. Function o is a vector of *output mapping functions* $[o_1, \ldots, o_k]$ where each of its components $o_j : M \rightarrow P$, $1 \leq j \leq k$, is an output mapping function which interfaces the output data structure and the process structure.

15. The notation "$\langle\ \rangle$" is a short-hand for component-wise function application of a vector of functions to another vector of functions (or elements), for instance,

$$\mathbf{F}\langle[\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m]\rangle \stackrel{\text{def}}{=} [F_1(\mathbf{u}_1), F_2(\mathbf{u}_2), \ldots, F_m(\mathbf{u}_m)],$$

$$\mathbf{X}\langle[\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_l]\rangle \stackrel{\text{def}}{=} [X_1(\mathbf{b}_1), X_2(\mathbf{b}_2), \ldots, X_l(\mathbf{b}_l)].$$

16. Functions $\tau_{\mathbf{V}}$ and $\phi_{\mathbf{V}}$ are continuous functions over the specified domains. Thus Equation (1) is a system of fixed-point equations, and on its right hand side,

$$\psi \stackrel{\text{def}}{=} \lambda\mathbf{F}.\lambda\mathbf{v}.\phi_{\mathbf{V}}(\mathbf{F}\langle\tau_{\mathbf{V}}(\mathbf{v})\rangle, \mathbf{X}\langle\iota(\mathbf{v})\rangle)$$

is a continuous function mapping from $E_1 \times \cdots \times E_m$ to $E_1 \times \cdots \times E_m$.

Definition 2.2 (Program Abstraction) Let the least fixed-point of ψ be denoted by $\mathbf{F}^{\infty}(\mathbf{X})$, and the notation $\circ$ stands for functional composition. The function $f_{\mathcal{P}} \stackrel{\text{def}}{=} \lambda\mathbf{m}.\lambda\mathbf{X}.\epsilon\circ\mathbf{F}^{\infty}(\mathbf{X})\langle o(\mathbf{m})\rangle$ is said to be the function *implemented by* the parallel program $\mathcal{P}$, where $\mathbf{m} \in M$, the output data structure, implies that each component $o_j(\mathbf{m})$ of $o(\mathbf{m})$ is in P, and ϵ is a projection from $D_1 \times D_2 \times \cdots \times D_m$. The value $f_{\mathcal{P}}(\mathbf{m})(\mathbf{X}_0)$ for any $\mathbf{m} \in M$ and any vector of input functions $\mathbf{X}_0$ is called a vector of outputs of program $\mathcal{P}$.

The need for the projection ϵ stems from the fact that sometimes a certain data stream F_i is only auxiliary and its values are not of interest, and hence the corresponding component in the vector of outputs is disregarded.
Remark: (Structured Programming) A **Crystal** program may contain nested levels of **Crystal** programs since any function used in the definition of a program $\mathcal{P}$, such as $\phi_{\mathbf{V}}$, can be implemented by another parallel program $\mathcal{P}'$.

Definition 2.3 (Program Equivalence) Two programs $\mathcal{P}$ and $\mathcal{P}'$ are *equivalent* if and only if $f_{\mathcal{P}} = f_{\mathcal{P}'}$, i.e., they implement the same function.

3 Definitions as Parallel Programs

Many familiar problem definitions with ensured correctness can be viewed as **Crystal** programs, and thus be interpreted to perform parallel computations. For example:

Program 1 (Integer Partition)

$$C(i,j) = \begin{cases} i = 1 \rightarrow 1 \\ i > 1 \rightarrow \begin{cases} i > j \rightarrow C(i-1,j) \\ i = j \rightarrow C(i-1,j) + 1 \\ i < j \rightarrow C(i-1,j) + C(i,j-i). \end{cases} \end{cases} \tag{2}$$

Its correctness can be verified easily by an inductive argument. It can be seen as a **Crystal** program

$$P_{int-par} = (\mathcal{N}^2, \{\mathcal{N}\}, \{C\}, \{\mathbf{1}\}, \mathcal{N}^2, \{\phi_{(i,j)}\}_{(i,j)\in\mathcal{N}^2}, \{\tau_{(i,j)}\}_{(i,j)\in\mathcal{N}^2}, \epsilon_2, \mathrm{Ide}),$$

where $\mathcal{N}$ is the set of positive integers; C is a functional variable; $\mathbf{1}$ is an input constant which is a function $\mathbf{1} : j \in \mathcal{N} \mapsto 1$. The local processing function $\phi_{(i,j)} = [\phi_{(i,j),1}, \phi_{(i,j),2}]$ where

$$\phi_{(i,j),q}(d_1,d_2,d_3) = \begin{cases} i = 1 \rightarrow d_3 \\ i > 1 \rightarrow \begin{cases} i > j \rightarrow d_1 \\ i = j \rightarrow d_1 + 1 \\ i < j \rightarrow d_1 + d_2 \end{cases} \end{cases} \quad \text{for } q = 1,2; \tag{3}$$

the communication function is

$$\tau_{(i,j)} = [\tau_{(i,j),1}, \tau_{(i,j),2}] = [(i-1,j),(i,j-i)];$$

the input mapping function $\epsilon_2 : (i,j) \mapsto j$ is a projection from $\mathcal{N}^2$ to $\mathcal{N}$ which chooses the second component from a pair of positive integers; the output mapping function is just an identity function $\mathrm{Ide} : (i,j) \mapsto (i,j)$ from $\mathcal{N}^2$ to $\mathcal{N}^2$, and the program body is

$$[C,C](i,j) = \phi_{(i,j)}([C,C]\langle\tau_{(i,j)}(i,j)\rangle, \mathbf{1}(\epsilon_2(i,j))), \quad \forall (i,j) \in \mathcal{N}^2. \tag{4}$$

This system of recursion equations is equivalent to Equation (3) since it defines a pair of functions $[C,C]$ in which two components are identical. The function $f_{P_{int-par}} = \lambda(i,j).\epsilon_1 \circ [C,C]^{\infty}(\mathrm{Ide}(i,j))$ is implemented by $P_{int-par}$, where $\epsilon_1 : (d_1,d_2) \mapsto d_1$ is a projection from $\mathcal{N}^2$ to $\mathcal{N}$, and $[C,C]^{\infty}$ is the least fixed point of Equation (4). Note that for this example, the three integers m, k, and l associated with a given **Crystal** program mentioned in Item 6 of Section 2 are 2, 1, and 1, respectively.

Program 2 (Matrix Multiplication)

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \times B(k,j) \text{ for } 1 \le i,j \le n. \tag{5}$$

This is the definition of multiplication of two real matrices A and B. The corresponding **Crystal** program consists of

$$P_{mm} = (\mathcal{N}^2, \{\Re\}, \{C\}, \{A,B\}, \mathcal{N}^2, \{\phi_{(i,j)}\}_{(i,j)\in\mathcal{N}^2}, \{\tau_{(i,j)}\}_{(i,j)\in\mathcal{N}^2}, [\alpha,\beta], \mathrm{Ide}),$$

where $\mathcal{N}$ is the set of positive integers $\{1,2,\ldots,n\}$, and $\Re$ is the type of the elements of the matrices. The local processing function $\phi_{(i,j)}$ is defined as

$$\phi_{(i,j)}(C, [a_{i1}, a_{i2}, \ldots, a_{in}, b_{1j}, b_{2j}, \ldots, b_{nj}]) \stackrel{\text{def}}{=} \sum_{k=1}^{n} a_{ik} \times b_{kj}.$$

Since the right hand side of Equation (5) contains only inputs, no communication function is necessary. The output mapping function is just the identity function Ide : $(i,j) \mapsto (i,j)$. The input mapping function $\iota = [\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n]$ where

$$\begin{aligned} \alpha_k(i,j) &= (i,k), \quad k = 1,2,\ldots,n, \\ \beta_k(i,j) &= (k,j), \quad k = 1,2,\ldots,n, \end{aligned}$$

and we let $\mathbf{X} = [A, \ldots, A, B, \ldots, B]$ be a vector of $2n$ components in which first n of them are A's and the last n of them are B's. Thus

$$\mathbf{X}\langle\iota(i,j)\rangle = [A(i,1), A(i,2), \ldots, A(i,n), B(1,j), B(2,j), \ldots, B(n,j)].$$

The program body

$$C(i,j) = \phi_{(i,j)}(C, \mathbf{X}\langle\iota(i,j)\rangle)$$

is exactly Equation (5). The function $f_{P_{mm}} = \lambda(i,j).C^{\infty}(i,j)$ is implemented by P_{mm}. Note that for this example, the three integers m, k, and l associated with a given **Crystal** program mentioned in Item 6 of Section 2 are 1, 1, and $2n$, respectively.

4 Ordering Parallel Processes

4.1 Process Structure

Definition 4.1 (Data Dependency) In a program P, a process $\mathbf{u} \in P$ *immediately precedes* a process $\mathbf{v} \in P$ ($\mathbf{u} \prec \mathbf{v}$), or $\mathbf{v}$ *immediately depends* on $\mathbf{u}$ ($\mathbf{v} \succ \mathbf{u}$) when $\mathbf{v}$ appears on the left-hand side of System (1) (the program body), and $\mathbf{u}$ appears on the right-hand side of the system as $F(\mathbf{u})$ where F is one of the data stream in the system. We also say that a process $\mathbf{v} \in P$ immediately depends on an input value ($\mathbf{x} \prec \mathbf{v}$) when some input $\mathbf{x}$ appears on the right-hand side of the system while $\mathbf{v}$ appears on the left-hand side.

Definition 4.2 (Source Processes) In a program P, if a process $\mathbf{v} \in P$ depends only on inputs, and not on any other process $\mathbf{u} \in P$, then $\mathbf{v}$ is called a source process, or simply, a source. Let Sr denote the set of source processes.

Definition 4.3 (Dependency Relation) A data dependency relation "precedes", denoted by "$\overset{*}{\prec}$", is the transitive closure of the relation "immediately precedes" ("$\prec$").

Definition 4.4 (Process Structure) $(P, \overset{*}{\prec})$ is called the process structure of a program P, and the program body (System (1)) must be such that the process structure is a well-founded set, and furthermore, every process can immediately depend on only a finite number of processes.

Remark: Process structure $(P, \overset{*}{\prec})$ can be drawn as a directed acyclic graph (DAG) with each node of the DAG being a process and an edge directed from a process $\mathbf{u}$ to a process $\mathbf{v}$ which is immediately dependent on $\mathbf{u}$. The finiteness of the number of dependent processes for each process says that the DAG must have finite fan-in degree at every node. The well-foundedness of $(P, \overset{*}{\prec})$ says that at any given node in the DAG, the chain of edges leading to the node must be of finite length. The entire DAG, however, may contain chains of infinite length.

Definition 4.5 (Data-independent Process Structure) Since every component of a communication function $\tau_{\mathbf{v},i}$ of program P does not depend on the value of any data stream $F(\mathbf{u})$ at any process, and it depends only on $\mathbf{v} \in P$, therefore $(P, \overset{*}{\prec})$ is called a *data-independent process structure.* In general, $\tau_{\mathbf{v},i}$ has a more complicated definition and $(P, \overset{*}{\prec})$ might be a data-dependent process structure.

Remark: Data-independent process structure is not necessarily static since it can be space-time variant. There is no necessity to incur any run-time overhead for mapping parallel processes to processors [2,3,5] for this class of programs since the mapping can be resolved at compile time.

Definition 4.6 (Partially Ordered Vector Space) Let V be a d-dimensional vector space over the rationals, for some non-negative integer d. A process structure $(P, \overset{*}{\prec})$ is said to be a (d-dimensional) partially ordered vector space if $P \subseteq V$. In this case, each process $\mathbf{v} \in P$ is also called a vector.

4.2 Wavefront Sequence

A wavefront sequence captures essentially the sequence of global state transitions of the ensemble of parallel processes of a program, and thus provides a useful tool for reasoning about the program.

Definition 4.7 Let $(W, <)$ be a well-ordered set, and $\widehat{0}$ be its least element. We say that $n' <_0 n$ (n' immediately less than n) if $n, n' \in W$, $n' < n$, and there exists no $m \in W$, $n' < m < n$. We also use the notation $n > n'$ if $n' < n$.

Definition 4.8 (Wavefront Sequence) A *wavefront sequence*

$$(w_i)_{i \in (W,<)}$$

of a program P is defined by (i) $w_{\widehat{0}} = Sr$, where $\widehat{0}$ is the least element of $(W, <)$, i.e., all sources belong to the first wavefront ($i = \widehat{0}$) of the sequence, and (ii) For $n > \widehat{0}$, $w_n \subseteq Q$, where

$$Q \stackrel{\text{def}}{=} \{\mathbf{v} : \forall \mathbf{u}, (\mathbf{u} \prec \mathbf{v} \Rightarrow \mathbf{u} \in w_k \text{ where } k < n) \text{ and } \mathbf{v} \notin w_i, \text{ where } j < n\},$$

i.e., a process may belong to wavefront at n if it does not belong to any previous wavefront at j, and all of its dependent processes belong to some previous wavefronts at k, $k < n$. We use the notation $(w_i)_{i \in (W,<)}(P)$ to denote the sequence of wavefronts defined by a program P.

Proposition 4.9 (Existence of a Wavefront Sequence) There exists a wavefront sequence for every program P.

Proof: Since the process structure $(P, \stackrel{*}{\prec})$ is a well-founded set with finite fan-in degree in the corresponding DAG, then the result of a topological sorting on $(P, \stackrel{*}{\prec})$ is a well-ordered set $(W, <)$. Then $(w_i)_{i \in (W,<)}$ with $w_i = \{i\}$ is a wavefront sequence. Note that W is no more than order type ω because of the finite fan-in degree in the DAG of $(P, \stackrel{*}{\prec})$. ∎

Definition 4.10 (Optimal Wavefront) A wavefront sequence $(o_i)_{i \in (W,<)}(P)$ is optimal if it is a wavefront sequence and for all $n > \widehat{0}$, $o_i = Q$, where Q is defined in Definition 4.8.

Proposition 4.11 If $(o_i)_{i \in (W,<)}(P)$ is an optimal wavefront sequence, then for all $n > \widehat{0}$:

$$\forall \mathbf{v} \in o_n, \exists \mathbf{u}, \mathbf{u} \prec \mathbf{v} \text{ such that } \mathbf{u} \in o_{n'}, \text{ where } n' <_0 n.$$

(In an optimal wavefront sequence, a process belongs to the wavefront at n if all of its dependent processes belong to wavefronts at $k < n$, and there is at least one dependent process belonging to wavefront at n' that is immediately before wavefront at n.)

The optimal sequence of wavefronts is crucial for devising the timing of a parallel implementation and for analysing the time complexity of a program. Finding a wavefront sequence for a program is easy, but ways of finding the optimal sequence of wavefronts may not be apparent. In Section 6, we illustrate how an optimal sequence can be obtained by an inductive procedure on a given wavefront sequence. Efficient parallel programs can then be derived based on the optimal sequence of wavefronts.

The wavefront sequence describes the proceeding of a computation independent of whether the parallel implementation of the program uses synchronous circuitry or an ensemble of asynchronous processors. Clearly, processes belonging to the same wavefront do not depend on one another. In an implementation, such processes could be arranged to execute simultaneously as in a synchronous system. However, such synchronization is not necessary; processes belonging to the same wavefront may be executed at different instances in real time (creating a "rippled" wavefront). Nor is it necessary for a process at wavefront w_i to be executed at a prior time than all of the processes at wavefront $w_{i'}$, $i < i'$. Such is the case in a self-timed system [15].

4.3 Complexity of the Naive Parallel Algorithm

In the matrix multiplication example, if we use one processor for each process in Program P_{mm}, we obtain immediately a naive parallel algorithm in which the local processing at each processor, and the communications between processors, are obtained from the definition as described in Section 2. Altogether $O(n^2)$ number of parallel processors are needed. However, the computation can be accomplished in a single step. This number of time steps is also exactly the total number of wavefronts in the optimal wavefront sequence, in this case consisting of only a single wavefront: $w_{\hat{0}}$ containing all processess (i, j), for $1 \leq i, j \leq n$, since all of them are source processes.

As exemplified here by the definition of matrix multiplication, when the constraints in the underlying hardware implementation are not taken into account in the computations and communications of a parallel program, the measure of its time complexity is too naive to be useful. In the next section, we show how to transform systematically a problem definition to a parallel program that does take into account realistic costs of hardware, and allows the number of wavefronts to serve as a valid measure of time complexity.

5 Program Transformations

The rules of transformations introduced below are motivated by the constraints imposed by hardware. These rules are extremely useful and apply to a large number of problems [2].

5.1 Problems with Large Fan-in and Fan-out Degrees

The time complexity of a **Crystal** program seems to be able to be determined by the number of wavefronts in the optimal wavefront sequence. However, due to the inherent physical constraints imposed by the driving capability of communication channels, power consumption, heat dissipation, memory bandwidth, etc., the time it takes for a process to complete a communication is, unfortunately, not entirely independent of the number of destination processes to which data must be sent (the fan-out degree), nor of the total number of sources and processes from which data must be received (the fan-in degree). We call the amount of time for completing a single process (single source) to process (single destination) communication a *unit communication time*. The number of unit communication times incurred for a communication that has a large fan-out degree, say degree $O(n)$, may become as large as $O(n)$.

5.2 Locality of Communications

Locality of a communication is another factor that affects the amount of time it takes for the completion of a communication. The farther away a communication must travel, the greater the number of unit communication times it takes. Locality of communications is defined with respect to a given class of process structures. For instance, with respect to a process structure that is a partially ordered vector space, locality is defined as:

Definition 5.1 If the process structure $(P, \overset{*}{\prec})$ is a partially ordered vector space, then the *path length* between two processes $\mathbf{v} \overset{\text{def}}{=} (v_1, \ldots, v_q) \in P$ and $\mathbf{u} \overset{\text{def}}{=} (u_1, \ldots, u_q) \in P$ with respect to P is defined to be $|v_1 - u_1| + \cdots + |v_q - u_q|$, where each component u_i and v_i for $i = 1, 2, \ldots, q$ are integers.

Given the definition of path length with respect to each class of process structures, locality can be defined:

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

Definition 5.3 The *order* of a system of recursion equations (the program body) is defined to be the maximum path length with respect to P over all pairs of processes $\mathbf{u}$ and $\mathbf{v}$ in P, where $\mathbf{u} \prec \mathbf{v}$. We call a system n-th order recursion equations if its order is n. A system of n'th order recursion equations is of *bounded order* if n is bounded.

5.3 Measuring Time Complexity at the Abstract Level

From the above discussions, we see that the time complexity cannot be reflected by the number of wavefronts alone and therefore the time spent for each communication that is implicit in the program must also be taken into account. However, we really would rather eliminate such low level consideration from the process of designing parallel programs. Note that the wavefront number does serve as a measure of time complexity for a program with bounded order and bounded fan-in and fan-out degrees [3] since a constant number of unit communication times does not affect the order of the time complexity. Thus if we manage to transform a given program to one with bounded order and bounded fan-in and fan-out degrees[4], then the cost for implementing the unbounded order or unbounded fan-in and fan-out degrees in the original program will be rightfully reflected in the new program at an abstract level, and hence there is no longer any need to consider the communication time.

Thus the elimination of large fan-in and fan-out degrees and high order terms from a program allows us to ignore implicit cost at the implementation level and measure the cost of a parallel programs in a realistic way at an abstract level.

5.4 Counting Fan-in and Fan-out Degrees

Definition 5.4 The fan-in degree of a process $\mathbf{u}$ is the total number of distinct processes $\mathbf{v}$ and inputs $\mathbf{x}$ appearing on the right hand side of a system of equation(s) with $\mathbf{u}$ appearing on its left-hand side.

Conversely,

Definition 5.5 The fan-out degree of a process $\mathbf{u}$ (or an input $\mathbf{x}$) is the total number of times a process $\mathbf{v}$ appears on the left-hand side of a system of recursion equations while process $\mathbf{u}$ (or an input $\mathbf{x}$) appears on its right-hand side, each time a distinct $\mathbf{v}$.

Definition 5.6 The fan-in (or fan-out) degree of a system of recursion equation(s) is the maximum fan-in degree over all processes (and inputs $\mathbf{x}$ in the case of fan-out degree) defined by the system.

5.5 Reducing the Fan-in Degrees

A large fan-in degree comes from an associative function which has a large number of arguments. The transformation for reducing fan-in degree is that of replacing this function by another program which has a bounded fan-in degree that implements the associative function. As long as the replacing program implements the function correctly, the transformed program is equivalent to the original one.

[3] Depending on the underlying machine architecture, sometimes a logarithmic fan-in or fan-out degree is acceptable up to a certain limit.

[4] or logarithmic fan-in and fan-out degrees.

Definition 5.7 (Associative Operation) An operation "$\bigoplus$", where $y = \bigoplus_{u<l\leq v} x(l)$, is associative if there exists a binary operation $\oplus$, a function z of variable l, and a function Φ

$$\Phi(z,x,l,u,v,\oplus) \stackrel{\text{def}}{=} \begin{cases} l = u \rightarrow \text{Ide}_{\oplus} \\ u < l \leq v \rightarrow z(l-1) \oplus x(l) \end{cases}$$

such that

$$\begin{aligned} y &= z(v) \qquad \text{and} \\ z(l) &= \Phi(z,x,l,u,v,\oplus) \end{aligned} \tag{6}$$

where $\text{Ide}_{\oplus}$ is the identity of "$\oplus$", and $z(l)$ has fan-in degree $1 + deg(x(l))$ and fan-out degree 1, and where $deg(x(l))$ is the fan-in degree of $x(l)$, assumed to be bounded.

Thus the high fan-in degree of an n-ary associative operation can be reduced by serializing the computation as the composition of a sequence of binary operations. Such serialization can be generalized to using compositions of a sequence of k-ary operations where k is bounded.

5.6 Reducing the Fan-out Degrees

In the case of reducing the fan-out degree of a program, the "broadcasting" function which transmits a value directly to many processes is replaced by a particular implementation that transmits the value by a series of communications from one process to the next.

Definition 5.8 (Concurrent Assignment) Let l, u, and v be integers. A set of equations $\{F(l) = E_l(x) : u < l < v\}$, contains an n-concurrent assignment $\{z(l) = x : u < l < v\}$ where $n = v - u - 1$, if such a z exists and $F(l) = E_l(z(l))$ for $u < l < v$.

Proposition 5.9 (Serial Assignment) A $(v-u-1)$-concurrent assignment $\{z(l) = x : u < l < v\}$ is equivalent to the sequence of serial assignments defined by the recursion equation

$$z(l) = \Psi(z,x,l,u,v,w), \text{ where} \tag{7}$$

$$\Psi(z,x,l,u,v,w) \stackrel{\text{def}}{=} \begin{cases} l = w \rightarrow x \\ w < l < v \rightarrow z(l-1) \\ u < l < w \rightarrow z(l+1) \end{cases} \tag{8}$$

for some fixed w, $u \leq w \leq v$.

In Equation (8), fan-out degrees of x and $z(l)$ for $u < l < w$ and $w < l < v$ are all 1, and the fan-out degree of $z(w)$ equals 2 if $u < w < v$ and equals 1 if $w = u$ or $w = v$.

Proof: $\{z(l) = x : u < l < v\}$ is the least fixed point of Equation (7). ∎
Remark: The choice of w in Proposition 5.9 concerns the issue of the locality of the communication between value x and variable $z(w)$. When x is an input, we let $w = u$, or symmetrically, $w = v$. When x is some value $F'(g(l))$ for some F' and g that depends on the index l, let w be chosen so that $|g(l) - w|$ is the minimum over all choices of w.

5.7 Matrix Multiplication with Bounded Degrees

From the definition of matrix multiplication P_{mm}, the large fan-in and fan-out degrees are eliminated and an equivalent program $P_{mm'}$ is obtained. First, we note that Equation (5) contains an associative operator "$\sum$" with unbounded fan-in degree n. Applying Definition 5.7 to "$\sum$", we obtain the following equation of the form of Equation (6):

$$\begin{aligned} C(i,j) &= \dot{c}(i,j)(n) \qquad \text{and} \\ \dot{c}(i,j)(k) &= \Phi(\dot{c}(i,j), \lambda k.[A(i,k) \times B(k,j)], k, 0, n, +) \end{aligned}$$

Expanding the definition of function Φ, and let $c(i,j,k) = \dot{c}(i,j)(k)$, we obtain

$$C(i,j) = c(i,j,n) \qquad \text{and}$$
$$c(i,j,k) = \begin{cases} k = 0 \rightarrow C_0(i,j) \\ 0 < k \leq n \rightarrow c(i,j,k-1) + A(i,k) \times B(k,j) \end{cases} \qquad (9)$$
where $C_0(i,j) = 0$, the identity of "+".

Next, by Definition 5.8, Equation (9) contains a total of $2n$ (2 for each value of k, $0 < k \leq n$) n-concurrent assignments, since for each k,

$\{\tilde{a}(i,k)(j) = A(i,k) : 0 < j < n+1\}$ such that $\tilde{c}(i,k)(j) = E^a{}_j(\tilde{a}(i,k)(j))$, $0 < k \leq n$

$\{\bar{b}(k,j)(i) = B(k,j) : 0 < i < n+1\}$ such that $\bar{c}(k,j)(i) = E^b{}_i(\bar{b}(k,j)(i))$, $0 < k \leq n$

where

$\tilde{c}(i,k)(j) = \bar{c}(k,j)(i) = c(i,j,k)$

$E^a{}_j \stackrel{\text{def}}{=} \lambda\widehat{A}.\begin{cases} k = 0 \rightarrow C_0(i,j) \\ 0 < k \leq n \rightarrow c(i,j,k-1) + \widehat{A} \times B(k,j), \end{cases}$ and

$E^b{}_i \stackrel{\text{def}}{=} \lambda\widehat{B}.\begin{cases} k = 0 \rightarrow C_0(i,j) \\ 0 < k \leq n \rightarrow c(i,j,k-1) + A(i,k) \times \widehat{B}. \end{cases}$

Now applying function Ψ in Proposition 5.9 to appropriate arguments of Equation (9), two equations of the form of Equation (7) are obtained:

$$\tilde{a}(i,k)(j) = \Psi(\tilde{a}(i,k), A(i,k), j, 0, n+1, 0), \qquad \text{and}$$
$$\bar{b}(k,j)(i) = \Psi(\bar{b}(k,j), B(k,j), i, 0, n+1, 0).$$

Expanding the definition of function Ψ and substitute $a(i,j,k)$ for $\tilde{a}(i,k)(j)$, $c(i,j,k)$ for $\tilde{c}(i,k)(j)$, $b(i,j,k)$ for $\bar{b}(k,j)(i)$, and $c(i,j,k)$ for $\bar{c}(k,j)(i)$, we obtain

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

and

$$b(i,j,k) = \begin{cases} i = 0 \rightarrow B(k,j) \\ 0 < i < n+1 \rightarrow b(i-1,j,k) \end{cases} \qquad (10)$$

such that

$$c(i,j,k) = \begin{cases} k = 0 \rightarrow C_0(i,j) \\ 0 < k \leq n \rightarrow c(i,j,k-1) + a(i,j,k) \times b(i,j,k) \end{cases}$$

Remark: The order of the application of Definition 5.7 and Proposition 5.9 to Equation (5) can be interchanged.

Note that System (10) is of first order and has fan-in and fan-out degree three. It constitutes the body of the **Crystal** program $\mathcal{P}_{mm'}$ whose interpretation as a parallel computation can be obtained as illustrated in Section 3.

Proposition 5.10 Program $\mathcal{P}_{mm'}$ is equivalent to Program $\mathcal{P}_{mm}$.

Proof: By Definition 5.7 and Proposition 5.9.

6 Incorporating Pipelining by Space-time Mapping

A naive implementation of a program P could use one processor for each process; however, after the execution of a process, a processor would be sitting idle, which is a wasting of resources. In general, a system of recursion equations with bounded fan-in and fan-out degrees defined on a d-dimensional partially ordered vector space consisting of $O(n^d)$ number of processes needs only $O(n^{d-1})$ number of processors. In other words, each processor can be re-used by $O(n)$ number of processes. The space-time mapping procedure described below achieves the saving of $O(n)$ number of processors for Program $P_{mm'}$.

Space-time mapping allows the source program to be purely mathematical and devoid of any concern about space-time or process synchronization, yet a resulting program after transformation has its synchronization resolved and achieves optimal timing and resource utilization. When the process structure is data-independent, the task of mapping processes to processors can be accomplished at compile time, and hence no run-time overhead is incurred.

6.1 Process Synchronization

Definition 6.1 (Synchronization Equalities) Let g be an integer-valued function where $g(\mathbf{v}) = g_0$ for all source processes $\mathbf{v} \in Sr$ and g_0 is called the initial synchronization point. For each non-source process $\mathbf{v} \in (P - Sr)$ of a program P, suppose process $\mathbf{v}$ has k dependent processes $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k$ ($\mathbf{u}_j \prec \mathbf{v}$), and $z_{\mathbf{v},j}$ for $1 \leq j \leq k$ are k positive integers. We call a set of k equalities

$$\begin{aligned} g(\mathbf{v}) &= g(\mathbf{u}_1) + z_{\mathbf{v},1} \\ &= g(\mathbf{u}_2) + z_{\mathbf{v},2} \\ &= \cdots \\ &= g(\mathbf{u}_k) + z_{\mathbf{v},k} \end{aligned} \tag{11}$$

the *synchronization equalities* (SE's) at process $\mathbf{v}$.

Proposition 6.2 (Optimal Timing) For a given program P, there exists an integer-valued function g over P that satisfies the set of synchronization equalities (11) at $\mathbf{v}$ with k positive integers $z_{\mathbf{v},j}$, where $1 \leq j \leq k$, for every process $\mathbf{v} \in P - Sr$ of program P, and $g(\mathbf{v}) = g_0$ for all $\mathbf{v} \in Sr$, and g is called a *timing function* of P. If, furthermore, g is minimized over all timing functions g' with respect to the same initial synchronization point g_0, i.e., $g(\mathbf{v}) \leq g'(\mathbf{v})$ for all $\mathbf{v} \in P$, and $g(\mathbf{v}) = g'(\mathbf{v}) = g_0$ for all $\mathbf{v} \in Sr$, then g is the optimal timing function for P.

Theorem 6.3 If $(o_i)_{i \in (W,<)}(P)$ is the optimal wavefront sequence, then there exists a mapping $\varphi : W \rightarrow T$ to the set of integers defined as $\varphi(\widehat{0}) = g_0$, where $\widehat{0}$ is the least element of $(W, <)$ and g_0 is an integer, and $i <_0 i' \Rightarrow \varphi(i') = \varphi(i) + 1$. The function $g : P \rightarrow T$, defined by $g(\mathbf{v}) = \varphi(i)$ if $\mathbf{v} \in o_i$, is the optimal timing function of P.

Proof: Function φ exists because the well-ordered set W is no more than order type ω. Note that $g_0 = \varphi(\widehat{0})$ is the initial synchronization point since for any source process $\mathbf{v} \in Sr$, by the definition of optimal wavefront, $\mathbf{v} \in o_{\widehat{0}}$. It follows that $g(\mathbf{v}) = \varphi(\widehat{0}) = g_0$. For all $\mathbf{v} \in P - Sr$, $z_{\mathbf{v},j} \stackrel{\text{def}}{=} g(\mathbf{v}) - g(\mathbf{u}_j)$ for each $\mathbf{u}_j \prec \mathbf{v}$. We have $g(\mathbf{v}) - g(\mathbf{u}_j) > 0$ by the definitions of wavefront sequence and function φ, and therefore integer $z_{\mathbf{v},j}$ is positive. Hence g satisfies the SE's at every process $\mathbf{v} \in P$, and is a timing function of P.

To show that it is also optimal, we assume that, on the contrary, there exists some timing function g' such that $g'(\mathbf{v}) < g(\mathbf{v})$ for some $\mathbf{v} \in P - Sr$ and $g(\mathbf{v}) = g'(\mathbf{v})$ for $\mathbf{v} \in Sr$. Let $S \stackrel{\text{def}}{=} \{\mathbf{v} : g'(\mathbf{v}) < g(\mathbf{v}), \mathbf{v} \in (P - Sr)\}$, and choose a least element $\mathbf{v} \in (S, \stackrel{*}{\prec})$, i.e., for all $\mathbf{u} \in S$, either $\mathbf{v}$ and $\mathbf{u}$ are not related or $\mathbf{v} \stackrel{*}{\prec} \mathbf{u}$. Since $\mathbf{v}$ is a least element of S, $g(\mathbf{u}_j) = g'(\mathbf{u}_j)$ for any $\mathbf{u}_j \prec \mathbf{v}$. Since $(o_i)_{i \in (W,<)}$ is the optimal wavefront sequence, there exists $\mathbf{u}_j$, such that

$z_{\mathbf{v},j} = g(\mathbf{v}) - g(\mathbf{u}_j) = 1$. Since g' also is a timing function and satisfies the SE's at $\mathbf{v}$, there exists a positive integer z, such that $g'(\mathbf{v}) = g'(\mathbf{u}_j) + z$. Now $z = g'(\mathbf{v}) - g'(\mathbf{u}_j) < g(\mathbf{v}) - g(\mathbf{u}_j) = 1$ implies that z is not positive, which is a contradiction. Hence g is the optimal timing function. ∎

Theorem 6.4 If g is an optimal timing function, then $(o_i)_{i\in(W,<)}$ is an optimal wavefront sequence, where $o_i = \{\mathbf{v} : g(\mathbf{v}) = i\}$, and $W \stackrel{\text{def}}{=} \{g(\mathbf{v}) : \mathbf{v} \in P\}$.

Proof: Since (1) $g(\mathbf{v})$ is integer-valued, (2) $g(\mathbf{v}) = g_0 \leq g(\mathbf{u})$ for any $\mathbf{v} \in Sr$ and for all $\mathbf{u} \in P$, set $(W \stackrel{\text{def}}{=} \{g(\mathbf{v}) : \mathbf{v} \in P\}, <)$ is well-ordered with g_0 being the least element. Since $g(\mathbf{v}) < g(\mathbf{u})$ whenever $\mathbf{v} \prec \mathbf{u}$, $(o_i)_{i\in W}$ is a wavefront sequence.

Since g is optimal, referring to the SE's it satisfies, for every $\mathbf{v} \in P$, there exists some j, $1 \leq j \leq k$ such that $z_{\mathbf{v},j} = g(\mathbf{v}) - g(\mathbf{u}_j) = 1$, where $\mathbf{u}_j \prec \mathbf{v}$. Otherwise $z_{\mathbf{v},j}$ for all $1 \leq j \leq k$ can be decreased by an equal amount to obtain $z'_{\mathbf{v},j} < z_{\mathbf{v},j}$, resulting in a contradition that g is not optimal. For every $\mathbf{v} \in P$ and such $\mathbf{u}_j \prec \mathbf{v}$, if $\mathbf{v} \in o_i$ then $\mathbf{u}_j \in o_{i-1}$ because $g(\mathbf{u}_j) = g(\mathbf{v}) - 1 = i - 1$. Since $i - 1 <_0 i$, $(o_i)_{i\in W}$ indeed is the optimal wavefront sequence. ∎

Theorem 6.5 (Existence of Optimal Timing Function) If there exists a wavefront sequence $(w_i)_{i\in(W,<)}$ for a program P, then a unique optimal timing function g exists.

Proof: By Proposition 4.9, a wavefront sequence $(w_i)_{i\in(W,<)}$ exists for P. Timing function g can be constructed from $(w_i)_{i\in(W,<)}$ by the following algorithm:
Algorithm T:

1. Let $g(\mathbf{v}) = g_0$ for all sources $\mathbf{v} \in Sr$, where g_0 is an integer-valued constant.
2. Let the SE's at every process $\mathbf{v} \in (P - Sr)$ be solved according to the ordering: For each $i \in W$, $i > \widehat{0}$, in increasing order of i:
   (a) solve the SE's at every $\mathbf{v} \in w_i$ for some $k(\mathbf{v})$ unknown positive integer values $z_{\mathbf{v},j}$ for $1 \leq j \leq k(\mathbf{v})$ and $z_{\mathbf{v},j}$ is minimized.
   (b) assigning $g(\mathbf{v}) := g(\mathbf{u}_j) + z_{\mathbf{v},j}$, for some $j \in \{1, 2, \ldots, k(\mathbf{v})\}$.

By induction on $(w_i)_{i\in(W,<)}$, g is the optimal timing function. To show its uniqueness, let us suppose that there exists another function $g' \neq g$ which is also an optimal timing function. Then there exists some $\mathbf{v}$ such that $g(\mathbf{v}) \neq g'(\mathbf{v})$, and the function $f(\mathbf{v}) \stackrel{\text{def}}{=} \min(g(\mathbf{v}), g'(\mathbf{v}))$ instead of g or g' should be the optimal timing function, which is a contradiction. ∎

Corollary 6.6 (Existence of the Unique Optimal Wavefront Sequence) The unique optimal wavefront sequence $(o_i)_{i\in(W,<)}$ for any program P exists.

Proof: By theorem 6.5 the unique optimal timing function g can be constructed. By Theorem 6.4, $(o_i)_{i\in(W,<)}$ is the optimal wavefront sequence, where $o_i = \{\mathbf{v} : g(\mathbf{v}) = i\}$, and $W \stackrel{\text{def}}{=} \{g(\mathbf{v}) : \mathbf{v} \in P\}$. ∎

6.2 Mapping Processes to Processors

Let S denote a set of *processors*, and T be a subset of the set of non-negative integers. We call each execution of a process by a processor an *invocation* of the processor. Let $t \in T$ be a non-negative integer for labeling the invocations so that the processes executed in the same processor can be differentiated. Let each invocation of a processor be denoted by (s, t) where $s \in S$ is a processor. Let $S \times T$ denote the set of all invocations. The concept of *pipelining* and a *space-time mapping* can now be defined as follows:

Proposition 6.7 (Pipelining Function) Let R be a subset of the set of processes P. There exists a one-to-one function $h : R \rightarrow T$ that is strictly monotonic from $(R, \overset{*}{\prec})$ to $(T, <)$, i.e., $h(\mathbf{u}) < h(\mathbf{v})$ if $\mathbf{u} \overset{*}{\prec} \mathbf{v}$. We say that processes in R are *pipelined through* a processor s. The *pipelining factor* of a processor s is $|R|$, the cardinality of R, and h is called a pipelining function.

Definition 6.8 (Space-time Mapping) A space-time mapping $\mathbf{g} \overset{\text{def}}{=} (g_S, g_T)$ of a program $\mathcal{P}$ is a pair of functions $g_S : P \rightarrow S$ and $g_T : P \rightarrow T$ such that for every $R_{s \in S}$, where $R_s \overset{\text{def}}{=} \{\mathbf{v} : g_S(\mathbf{v}) = s\}$, all of the processes $\mathbf{v} \in R_s$ pipeline through s with pipeline function $g_T|R_s$, the restriction of g_T to $R_s \subseteq P$.

Proposition 6.9 A space-time mapping $\mathbf{g}$ is a one-to-one function, and its inverse $\mathbf{g}^{-1}$ is well-defined.

Definition 6.10 An optimal space-time mapping $\mathbf{g} = (g_S, g_T)$ of a program $\mathcal{P}$ is a space-time mapping such that g_T is the optimal timing function of $\mathcal{P}$.

Remark: An optimal space-time mapping $\mathbf{g} = (g_S, g_T)$ of a program $\mathcal{P}$ is not unique, in spite of the uniqueness of the optimal timing function g_T, because there could be many choices for g_S.
Remark: The method of space-time mapping suits particularly well the class of programs that have partially ordered vector spaces as process structures. For this class of programs, there exists a method for finding function g_S in a systematic manner, which is dealt with elsewhere.
Remark: The efficiency of a program can be determined by its optimal timing function or optimal wavefront sequence. The more elements each wavefront contains gives rise to more parallelism *and* fewer number of wavefronts *and* better time complexity for the program. A more refined comparison of the efficiency of each of the pipelined programs derived from the same original program can be made by measures such as the pipelining factor.

Definition 6.11 (Optimal Pipelined Program) Given a program

$$\mathcal{P} = (P, \mathcal{D}, \mathcal{F}, \mathcal{X}, M, \{\phi_{\mathbf{v}}\}_{\mathbf{v} \in P}, \{\tau_{\mathbf{v}}\}_{\mathbf{v} \in P}, \iota, o)$$

and its optimal space-time mapping $\mathbf{g} : P \rightarrow S \times T$, a derived program

$$\hat{\mathcal{P}} = (S \times T, \mathcal{D}, \hat{\mathcal{F}}, \mathcal{X}, M, \{\hat{\phi}_{(s,t)}\}_{(s,t) \in S \times T}, \{\hat{\tau}_{(s,t)}\}_{(s,t) \in S \times T}, \hat{\iota}, \hat{o})$$

$$\hat{\mathbf{F}}(s,t) = \hat{\phi}_{(s,t)}(\hat{\mathbf{F}}\langle\hat{\tau}_{(s,t)}(s,t)\rangle, \mathbf{X}\langle\hat{\iota}(s,t)\rangle), \ \forall (s,t) \in S \times T, \tag{12}$$

where

$$\begin{aligned}
&\hat{\mathbf{F}} \overset{\text{def}}{=} \mathbf{F} \circ \mathbf{g}^{-1} \\
&\hat{\phi}_{(s,t)} = \lambda \mathbf{v}.\phi_{\mathbf{v}}(\mathbf{g}^{-1}(s,t)) \\
&\hat{\tau}_{(s,t)}(s,t) = \lambda \mathbf{v}.\tau_{\mathbf{v}}(\mathbf{g}^{-1}(s,t)) \\
&\hat{\iota} \overset{\text{def}}{=} \iota \circ \mathbf{g}^{-1} \\
&\hat{o} \overset{\text{def}}{=} o \circ \mathbf{g}^{-1}
\end{aligned}$$

is called an optimal *pipelined program* with respect to $\mathcal{P}$ by $\mathbf{g}$.

Proposition 6.12 Program $\hat{\mathcal{P}}$ is equivalent to program $\mathcal{P}$.

Proof: $\mathbf{v} = (\mathbf{g}^{-1} \circ \mathbf{g})(\mathbf{v}) = \mathbf{g}^{-1}(s,t)$. ∎

6.3 Space-time Mapping for Matrix Multiplications

The following is an example of synthesizing several efficient parallel programs from the bounded order and bounded degree matrix multiplication program $P_{mm'}$. Four space-time mappings $\mathbf{g}_1, \ldots, \mathbf{g}_4$ are constructed for $P_{mm'}$. An optimal pipelined program $\hat{P}_{mm_i}$ with respect to $P_{mm'}$ by $\mathbf{g}_i$, for each i, $i = 1, 2, 3, 4$, is obtained.

Proposition 6.13 Function $g(i,j,k) = i + j + k$ is the optimal timing function of program $P_{mm'}$.

Proof: Let

$$z_{(i,j,k),(0,0,1)} = z_{(i,j,k),(1,0,0)} = z_{(i,j,k),(0,1,0)} = 1 \tag{13}$$

First we show that g is the optimal timing function by showing that it satisfies the SE's at every (i,j,k) with positive integers defined in Equation (13).

$$\begin{aligned} & g(i,j,k) \\ &= g(i,j,k-1) + z_{(i,j,k),(0,0,1)} \\ &= i + j + (k-1) + 1 \\ &= g(i-1,j,k) + z_{(i,j,k),(1,0,0)} \\ &= (i-1) + j + k + 1 \\ &= g(i,j-1,k) + z_{(i,j,k),(0,1,0)} \\ &= i + (j-1) + k + 1 \\ &= i + j + k \end{aligned}$$

Note that if $k = 0$, the term associated with $k - 1$ in the equality disappears, and likewise for the cases $i = 0$ and $j = 0$. It is optimal because any positive integer $z_{(i,j,k),\mathbf{d}}$ assumes the least possible value for all (i,j,k) and any $\mathbf{d} \in \{(0,1,0),(1,0,0),(0,0,1)\}$. ∎

Remark: By Proposition 6.5, the timing function g can be obtained constructively.

Proposition 6.14 Let g_{s1} be defined as $g_{s1}(i,j,k) = (x,y) = (i,j)$, and g be the optimal timing function in Proposition 6.13. Then $\mathbf{g}_1 \stackrel{\text{def}}{=} [g_{s1}, g]$ is an optimal space-time mapping of program $P_{mm'}$.

Proof: Let $R_{(i,j)} \stackrel{\text{def}}{=} \{(i,j,k) : (i,j,k) \in P_{mm'}\}$ for any given (i,j), where $(P_{mm'}, \stackrel{*}{\prec})$ is the process structure of $P_{mm'}$. The function $g(i,j,k)|R_{(i,j)}$ is one-to-one because whenever $k \neq k'$, $g(i,j,k) \neq g(i,j,k')$. Furthermore, if $(i,j,k) \prec (i,j,k')$, then $g(i,j,k) < g(i,j,k')$, so $g|R_{(i,j)}$ (the restriction of g to $R_{(i,j)}$) is a strictly monotonic function. Thus $(i,j,k) \in R_{(i,j)}$ pipeline through (i,j) with pipeline function $g|R_{(i,j)}$. Hence $\mathbf{g}_1$ is a space-time mapping. It is the optimal mapping because g is the optimal timing function. ∎

Proposition 6.15 Let g_{s2} be defined as $g_{s2}(i,j,k) = (x,y) = (i+k, j+k)$, and g be the optimal timing function in Proposition 6.13. Then $\mathbf{g}_2 \stackrel{\text{def}}{=} [g_{s2}, g]$ is an optimal space-time mapping of program $P_{mm'}$.

Proof: Let $R_{(x,y)} \stackrel{\text{def}}{=} \{(x,y) : (x,y) = (i+k, j+k), (i,j,k) \in P_{mm'}\}$, where $(P_{mm'}, \stackrel{*}{\prec})$ is the process structure of $P_{mm'}$. The function $g(i,j,k)|R_{(x,y)}$ is one-to-one because whenever two different processes (i,j,k) and (i',j',k') are mapped to the same processor (x,y), i.e., $x = i + k = i' + k'$, $y = j + k = j' + k'$, and $(i,j,k) \neq (i',j',k')$, then $g(i,j,k)|R_{(x,y)} = (i+k) + (j+k) - k = x + y - k \neq x + y - k' = g(i',j',k')|R_{(x,y)}$. Furthermore, if two different processes are mapped to the same processor (x,y), then $(i,j,k) \stackrel{*}{\not\prec} (i',j',k')$ and

$(i',j',k') \overset{*}{\not\prec} (i,j,k)$ because whenever $i < i'$ then $k > k'$ (or $i > i'$ then $k < k'$) and similarly whenever $j < j'$ then $k > k'$ (or $j > j'$ then $k < k'$). Consequently, $g|R_{(x,y)}$ is trivially a strictly monotonic function, and thus $(i,j,k) \in R_{(x,y)}$ pipeline through processor $(x,y) = (i+k, j+k)$ with pipeline function $g|R_{(x,y)}$. Hence $\mathbf{g_2}$ is a space-time mapping. It is the optimal mapping because g is the optimal timing function. ∎

Proposition 6.16 Let g_{s3} be defined as $g_{s3}(i,j,k) = (x,y) = (i-k, j-k)$, and g be the optimal timing function in Proposition 6.13. Then $\mathbf{g_3} \overset{\text{def}}{=} [g_{s3}, g]$ is an optimal space-time mapping of program $P_{mm'}$.

Proof: Similar to Proposition 6.15. ∎

Proposition 6.17 Let g_{s4} be defined as $g_{s4}(i,j,k) = (x,y) = (i-j+k, k)$, g be the optimal timing function in Proposition 6.13. Then $\mathbf{g_4} \overset{\text{def}}{=} [g_{s4}, g]$ is an optimal space-time mapping of program $P_{mm'}$.

Proof: Let $R_{(x,y)} \overset{\text{def}}{=} \{(x,y) : (x,y) = (i-j+k, k), (i,j,k) \in P_{mm'}\}$, where $(P_{mm'}, \overset{*}{\prec})$ is the process structure of $P_{mm'}$. If two different processes (i,j,k) and (i',j',k') are mapped to the same processor $(x,y) = (i-j+k, k)$, then $k = k'$ and whenever $i < i'$ then $j < j'$, and hence $(i,j,k) \overset{*}{\prec} (i',j',k')$ (similar for the case $i > i'$, which implies $j > j'$). Therefore $g|R_{(x,y)}$ is strictly monotonic, and $g(i,j,k) \neq g(i',j',k')$ and consequently $g|R_{(x,y)}$ is one-to-one. ∎

Figure 2 illustrates the process structure P in the original coordinate system with indices (i,j,k) and in the new coordinate systems with indices (x_i, y_i, t) where $(x_i, y_i, t) = \mathbf{g}_i(i,j,k)$ for each space-time mapping $\mathbf{g}_i$, $i = 1,2,3,4$.

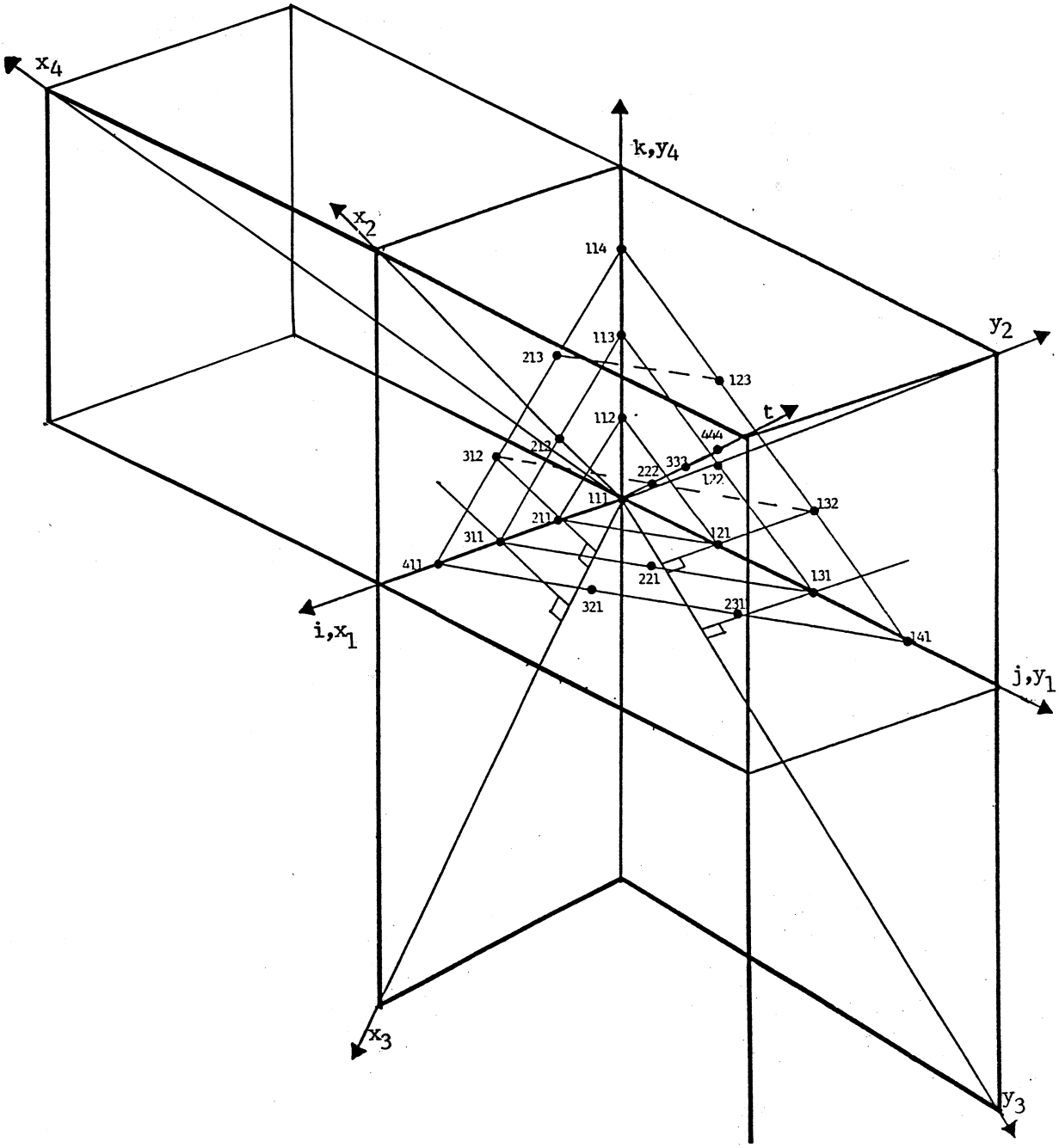

Figure 2: Process structure in the original coordinate system, and the new axis x_i, y_i, and t after space-time mapping $\mathbf{g}_i$ for $i = 1, 2, 3, 4$.

7 Derivation and Comparison of Target Programs

7.1 Target Pipelined Programs

Having obtained the space-time mapping $\mathbf{g}_i$, $i = 1, 2, 3, 4$ for program $P_{mm'}$, target pipelined programs $\hat{P}_{mm_i}$ can be obtained as described in Definition 6.11. Explicitly, the transformation is performed as follows: Taking $\mathbf{g}_2$ as an example, the target program $\hat{P}_{mm_2}$ is derived by using the following identity:

Proposition 7.1 Let $\hat{c} = c\mathbf{g}_2^{-1}$, where $\mathbf{g}_2^{-1}$ is the inverse of $\mathbf{g}_2$ as defined in Proposition 6.15. Then

$$c(i,j,k) = \hat{c}(x,y,t), \text{ and } c(i,j,k-1) = \hat{c}(x-1,y-1,t-1) \text{ for all } (i,j,k) \in P. \tag{14}$$

Proof: By definitions of $\mathbf{g}_2$ and $\hat{c}$, and functional composition,

$$c(i,j,k) = c(\mathbf{g}_2^{-1}\mathbf{g}_2(i,j,k)) = c(\mathbf{g}_2^{-1}(i+k, j+k, i+j+k)) = c(\mathbf{g}_2^{-1}(x,y,t)) = c\mathbf{g}_2^{-1}(x,y,t) = \hat{c}(x,y,t).$$

Similarly,

$$c(i,j,k-1) = c\mathbf{g}_2^{-1}(i+(k-1), j+(k-1), i+j+(k-1)) = c\mathbf{g}_2^{-1}(x-1,y-1,t-1)$$
$$= \hat{c}(x-1,y-1,t-1). \blacksquare$$

Similarly,

Proposition 7.2

$$a(i,j,k) = \hat{a}(x,y,t), \text{ and } a(i,j-1,k) = \hat{a}(x,y-1,t-1), \tag{15}$$

$$b(i,j,k) = \hat{b}(x,y,t), \text{ and } b(i-1,j,k) = \hat{a}(x-1,y,t-1). \tag{16}$$

Proposition 7.3 The following identities hold under space-time mapping $(x,y,t) = \mathbf{g}_2(i,j,k) = (i+k, j+k, i+j+k)$:

$$A(i,k) = A(t-y, x+y-t),\ B(k,j) = B(x+y-t, t-x),\ C(i,j) = C(t-y, t-x). \tag{17}$$

Proposition 7.4 The following identities between predicates hold under space-time mapping $(x,y,t) = \mathbf{g}_2(i,j,k) = (i+k, j+k, i+j+k)$:

$$\begin{aligned} &(j=0) = (t=x), (0<j\le n) = (x<t\le n+x) \\ &(i=0) = (t=y), (0<i\le n) = (y<t\le n+y) \\ &(k=0) = (t=x+y), (0<k\le n) = (x+y-n<t\le x+y). \end{aligned} \tag{18}$$

Substituting the identities $(i < n+1) = (i \le n)$, $(j < n+1) = (j \le n)$, and Identities (14), (15), (16), (17), and (18) into System (10), the program body of $P_{mm'}$, we obtain the body of the new pipelined program P_{mm_2}:

$$\begin{aligned} \hat{a}(x,y,t) &= \begin{cases} t = x \rightarrow A(x-y, y) \\ x < t \le n+x \rightarrow \hat{a}(x, y-1, t-1) \end{cases} \\ \hat{b}(x,y,t) &= \begin{cases} t = y \rightarrow B(x, y-x) \\ y < t \le n+y \rightarrow \hat{b}(x-1, y, t-1) \end{cases} \\ \hat{c}(x,y,t) &= \begin{cases} t = x+y \rightarrow C_0(x,y) \\ x+y-n < t \le x+y \rightarrow \hat{c}(x-1,y-1,t-1) + \hat{a}(x,y,t) \times \hat{b}(x,y,t). \end{cases} \end{aligned} \tag{19}$$

Proposition 7.5 The set of processes $\hat{P}_{mm_2}$ of Program $\hat{P}_{mm_2}$ is

$$\hat{P}_{mm_2} = \{(x,y,t) : (x,y,t) \in Q, -(n-1) \leq x-y \leq (n-1),\ (x+y) \leq t \leq n+(x+y)\},$$

a subset of the index set $Q \stackrel{\text{def}}{=} \{x : 1 \leq x \leq 2n\} \times \{y : 1 \leq y \leq 2n\} \times \{t : 2 \leq t \leq 3n\}$.

Proof: $(x,y,t) = \mathbf{g}_2(i,j,k)$ ∎

7.2 Initialization of a Program

For practical reasons, it is often preferable to initialize a parallel program at some initial time $t = t_0$ for all processes at once rather than allow the initialization to be space-time variant as described by the predicates $t = x$, $t = y$, and $t = x + y$ in System (19). Such initialization can be achieved by program transformations in which an input assigned to an invocation (x,y,t) with $t > t_0$ is moved in space and time by translation to another process (x',y',t_0).

In the following, we choose $t_0 = 0$.

Proposition 7.6 Recursion equation

$$\hat{a}(x,y,t) = \begin{cases} t = 0 \rightarrow A(-y, x+y) \\ 0 < t \leq x \rightarrow \hat{a}(x, y-1, t-1) \end{cases} \tag{20}$$

implies

$$\hat{a}(x,y,x) = A(x-y,y). \tag{21}$$

Proof: By Equation (20),

$$\begin{aligned} &\hat{a}(x,y,0) = A(-y,x+y) \text{ by the case } t = 0 \text{ of Equation (20)} \\ &= \hat{a}(x,y+x,x) \text{ by the case } t > 0 \text{ of Equation (20).} \end{aligned}$$

Let $y' = y+x$, then $\hat{a}(x,y',x) = A(-(y'-x), x+(y'-x)) = A(x-y', y')$ which is just Equation (21) with y' as a formal parameter instead of y. ∎

Similarly,

Proposition 7.7 Recursion equation

$$\hat{b}(x,y,t) = \begin{cases} t = 0 \rightarrow B(x+y, -x) \\ 0 < t \leq y \rightarrow \hat{b}(x-1, y, t-1) \end{cases} \quad \text{implies } \hat{b}(x,y,y) = B(x, y-x). \tag{22}$$

Proposition 7.8 Recursion equation

$$\hat{c}(x,y,t) = \begin{cases} t = 0 \rightarrow C_0(-y,-x) \\ 0 < t \leq x+y \rightarrow \hat{c}(x-1, y-1, t-1) \end{cases} \quad \text{implies } \hat{c}(x,y,x+y) = C_0(x,y). \tag{23}$$

Incorporating the above propositions into System (19), we obtain the program body of $P_{mm_2'}$.

$$\begin{aligned} \hat{a}(x,y,t) &= \begin{cases} t = 0 \rightarrow A(-y, x+y) \\ 0 < t \leq n+x \rightarrow \hat{a}(x, y-1, t-1) \end{cases} \\ \hat{b}(x,y,t) &= \begin{cases} t = 0 \rightarrow B(x+y, -x) \\ 0 < t \leq n+y \rightarrow \hat{b}(x-1, y, t-1) \end{cases} \\ \hat{c}(x,y,t) &= \begin{cases} t = 0 \rightarrow C_0(-y,-x) \\ 0 < t \leq x+y \rightarrow \hat{c}(x-1, y-1, t-1) \\ x+y-n < t \leq x+y \rightarrow \hat{c}(x-1, y-1, t-1) + \hat{a}(x,y,t) \times \hat{b}(x,y,t). \end{cases} \end{aligned} \tag{24}$$

Note that Equation (24) implies Equation (19) but not the other way around, which says that the set of processes $\hat{P}_{mm_2}$ of the original program is a subset of the set of process $\hat{P}_{mm'_2}$ of the new program. The function each of them implements can be made equivalent if the vector of outputs of the new program are those obtained from elements $\mathbf{m}$ of the output data structure M such that $o(\mathbf{m})$ are in the original set of processes $\hat{P}_{mm_2}$ only.

From Equation (24), various parts of the body of Program $\hat{P}_{mm'_2}$, such as the local processing function, communication function, etc., can be obtained as illustrated in Section 3, which can then be used for either generating multiprocessor object code or translating to an architectural level specification for direct VLSI implementations. Figure 3 illustrates the projection of the process structure $\hat{P}_{mm'_2}$ defined by Program $\hat{P}_{mm'_2}$ to the plane $t = 0$. This projection results in the processors configured as an array of processors [16] shown in the figure, where arrows indicating the direction of the data flow, and the initialization of the input data, are also shown. Note that set $\hat{P}_{mm_2}$ defined in Proposition 7.5 is a subset of $\hat{P}_{mm'_2}$ and its projection to the plane $t = 0$ tells where all the processors are located. Similarly, from space-time mapping $\mathbf{g}_3$, a different Program $\hat{P}_{mm'_3}$ [8] can be obtained and its implementation diagram is shown in Figure 4. Readers are encouraged to derive Programs $\hat{P}_{mm'_1}$ and $\hat{P}_{mm'_4}$ and draw diagrams similar to Figures 3 and 4.

7.3 Comparison of Target Programs

We now discuss some features that distinguish the resulting parallel implementations. The two implementations compared are Program $\hat{P}_{mm'_2}$ by space-time mapping $\mathbf{g}_2$ and Program $\hat{P}_{mm'_3}$ by space-time mapping $\mathbf{g}_3$.

7.3.1 Pipelining Factor

Now consider both Program $\hat{P}_{mm'_2}$ and Program $\hat{P}_{mm'_3}$ each as a black box (a processor at the next higher level), and each of the input matrices A, B, and C_0 on the whole as a "packet" of inputs. Suppose there are a large number of matrix multiplications that need to be performed. They can be pipelined through either Program $\hat{P}_{mm'_2}$ or Program $\hat{P}_{mm'_3}$. For this purpose, do these two programs have the same efficiency?

What is noteworthy is the initialization of inputs as shown in Figures 3 and 4. The input matrix element $C_0(n, n)$ is n units away in the diagonal direction from $C_0(1, 1)$ in Program $\hat{P}_{mm'_2}$; however, it is $3n$ units away from $C_0(1, 1)$ in Program $\hat{P}_{mm'_3}$. A similar situation holds for $A(n, n)$ and $B(n, n)$. Consequently, in a fixed number of steps — say, $k \cdot n$ steps — there can be k packets of inputs taken by Program $\hat{P}_{mm'_2}$ in each direction, whereas only $\frac{k}{3}$ packets of inputs can be taken by Program $\hat{P}_{mm'_3}$. Hence the pipelining factor of Program $\hat{P}_{mm'_2}$ is k whereas that of Program $\hat{P}_{mm'_3}$ is $\frac{k}{3}$.

We can conclude that when a large number of matrix multiplications must be preformed by each program, Program $\hat{P}_{mm'_2}$ is 3 times more efficient than Program $\hat{P}_{mm'_3}$[5], in spite of the fact that they both perform a single matrix multiplication in $3n$ steps.

7.3.2 Processor Utilization

Another indicator of the efficiency of a program is how often each processor is utilized during a computation. To find out if each processor (x, y) is active at each of its invocations (x, y, t), the inverse space-time mappings $\mathbf{g}_2^{-1}$ and $\mathbf{g}_3^{-1}$ are used:

[5]The loss in efficiency of this program by a factor of three can be recovered if three input matrices are interleaved in each packet of inputs. The correctness of such interleaving can again be verified algebraically as illustrated in this paper.

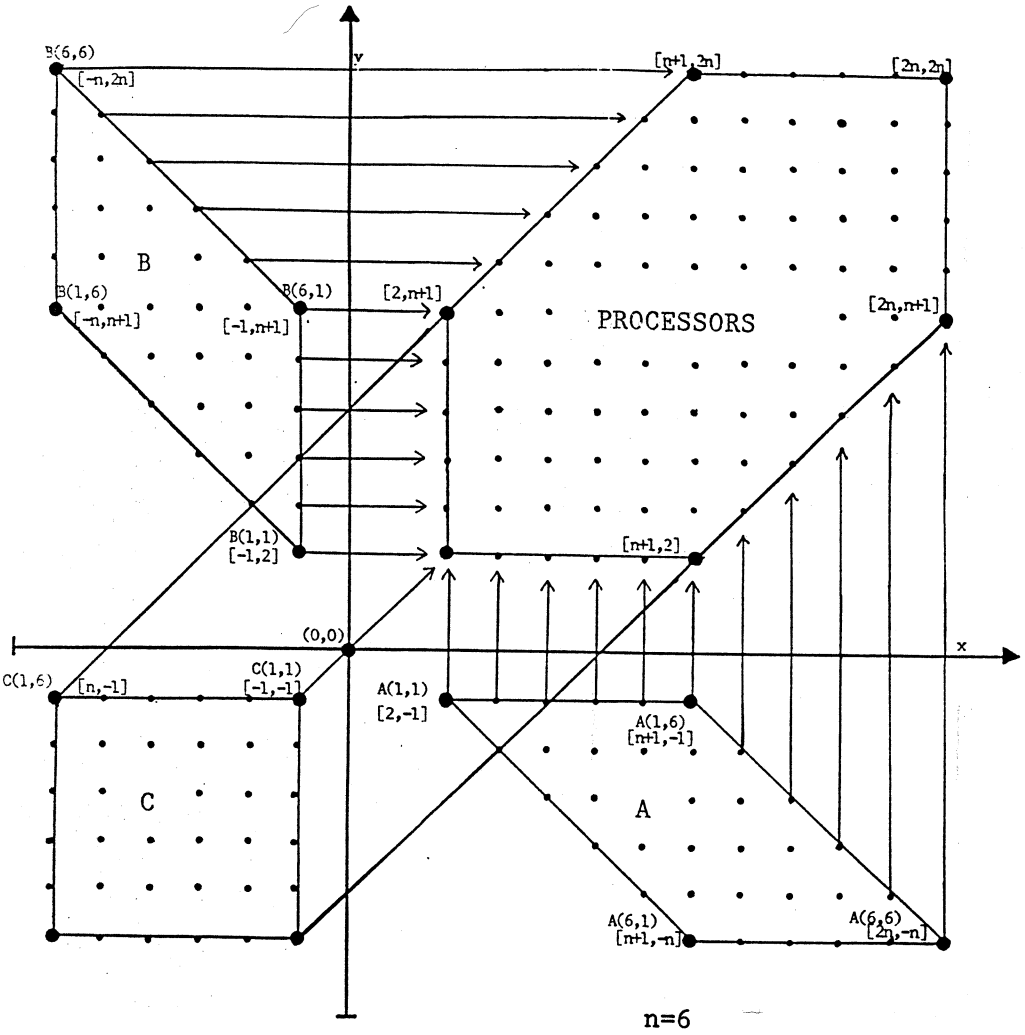

Figure 3: The systolic network and inputs using space-time mapping $\mathbf{g}_2$.

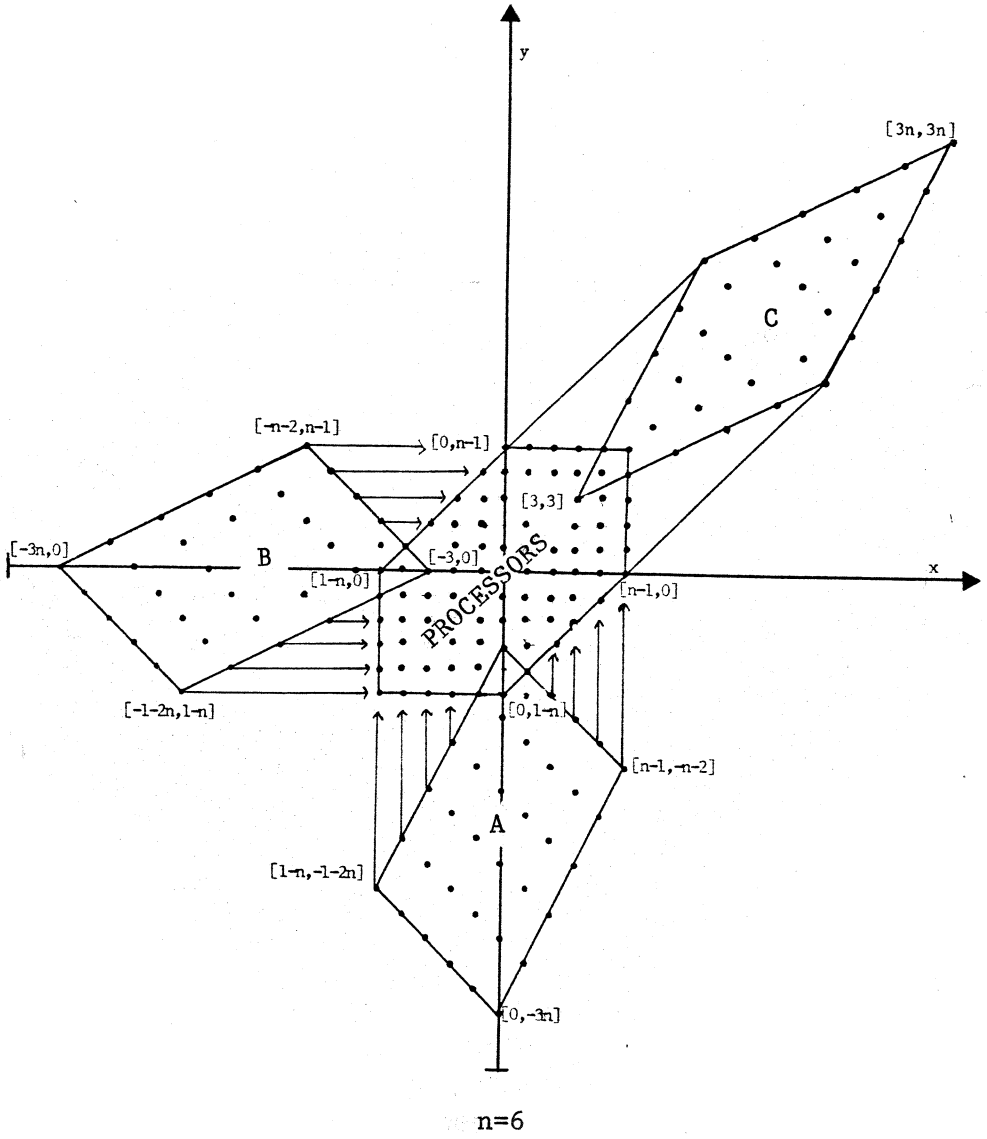

Figure 4: The systolic network and inputs using space-time mapping g_3.

Proposition 7.9 In Program $\hat{P}_{mm'_2}$, a processor (x, y) which executes process (i, j, k) at time t will execute $(i+1, j+1, k-1)$ at time $t+1$ if $(i+1, j+1, k-1) \in \hat{P}_{mm'_2}$.

Proposition 7.10 In Program $\hat{P}_{mm'_3}$, a processor (x, y) which executes process (i, j, k) at time t will execute $(i+1, j+1, k+1)$ at time $t+3$ if $(i+1, j+1, k+1) \in \hat{P}_{mm'_3}$. No process is executed at $(x, y, t+1)$ or $(x, y, t+2)$ in processor (x, y).

By comparison, each processor is busy at every time step in Program $\hat{P}_{mm'_2}$, but only busy one out of every three time steps in Program $\hat{P}_{mm'_3}$. This comparison of processor utilization agrees with the above comparison of pipelining factors.

Comparisons made among the different pipelined programs illustrate one of the most interesting facets — the multi-dimensional programming space — of parallel programming which does not have a counterpart in sequential programming.

8 Concluding Remarks

In this paper, we have shown that efficient parallel programs can be synthesized by a series of mathematical tranformations with guaranteed correctness. The mathematics used are, in fact, quite conventional and straightforward. It is interesting, perhaps even surprising, to see that the approach of taking straightforward mathematical definitions as initial specifications could yield such powerful results in synthesizing efficient parallel programs.

As we recall, very often a sequential program over-specifies a problem in the sense that extraneous data dependency is introduced due to the constraint imposed by the sequential model of computation. A mathematical definition of a problem, on the other hand, is a relatively "pure" description strictly for the purpose of commmunicating the ideas it embodies; it is void of any extraneous dependency. Because parallelism is what needs to be exploited in order to gain high performances, and because fewer dependencies imply more parallelism, it might therefore not be a coincidence that a simple mathematical definition can serve so well as an initial specification of a parallel program.

The efficiency of a naive parallel program, which is obtained from the problem definition by interpretation, is improved by taking into account the constraints imposed by the hardware technology. For instance, bounded fan-in and fan-out degrees and locality of communications are realistic constraints imposed by parallel machines and the VLSI technology. They can be formulated mathematically and incorporated into programs by algebraic transformations. The result is a realistic parallel program with improved efficiency.

Another interesting aspect of a parallel computation, which does not have a counterpart in the sequential case, is that of the many design possibilities in a multi-dimensional space of processors and time, as demonstrated by the two final parallel programs. Now the 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 has different properties and may be suited for different purposes.

This property of the multi-dimensional design space of parallel programs has a more fundamental implication: modeling a parallel computation as a composition of sequential processes, where each sequential process is modeled by a sequence of events (as in CSP), is somewhat biased, and may prevent a truly high-level view of a parallel system. Such modeling imposes an asymmetry in space and time because it presupposes a sequential process (consisting of events occurring through a sequence of time steps in a particular point in space) as being an interesting logical unit. Many interesting parallel computations, however, have as a logical unit a sequence of events "moving" in both space and time, whereas all events in a given sequential

process are, in fact, unrelated, as with those illustrated by two systolic matrix multiplication programs above.

Once the prejudice against the new element in parallel computations — the space component — is eradicated, an event can then be modeled as a purely mathematical object, because the very symmetry of space and time allows us to disregard them totally while programming. To develop an implementation of a parallel program, a space-time mapping of events then ensues, and a single program may yield many different implementations. On the contrary, when the sequential process model is used while programming, a different program must be written for each different implementation. Thus by taking a symmetrical view of both space and time in **Crystal**, the programming effort is elevated to a level that is beyond a particular space-time implementation of a program.

Another implication we can draw from this work on systolic computation is that efficient task decomposition and distribution to parallel processes may not be as hard as it seems. This certainly is encouraging for us in investigating programming methodologies for various other classes of computations.

References

[1] K. Apt, N. Francez, and W. De Roever. A proof system for communicating sequential processes. *ACM Trans. Prog. Lang. Syst.*, 3(2):359–385, July 1980.

[2] M. C. Chen. Crystal: a synthesis approach to programming parallel machines. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, August 1985.

[3] M. C. Chen. The generation of a class of multipliers: a synthesis approach to the design of highly parallel algorithms in vlsi. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 116–121, October 1985.

[4] M. C. Chen. *Space-time Algorithms: Semantics and Methodology.* PhD thesis, California Institute of Technology, May 1983.

[5] M. C. Chen. Synthesizing systolic designs. In *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, pages 209–215, May 1985.

[6] M. C. Chen. Transformations of parallel programs in crystal. In *The Proceedings of the IFIP 86, Dublin, Ireland*, September 1986.

[7] C. A. R. Hoare. A calculus of total correctness for communicating processes. *SCi. Comput. Prog. 1.*, ():48–72, 1981.

[8] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI Processor Arrays*, chapter 8.3. Addison-Wesley, 1980.

[9] H.T. Kung. Why systolic architecture? *IEEE Computer*, :37–46, January 1982.

[10] L. Lamport and F. B. Schneider. The 'Hoare logic' of CSP, and all that. *ACM Tras. Prog. Lang. Syst.*, (6):281–296, April 1984.

[11] G. Levin and D. Gries. Proof techniques for communicating sequential processes. *Acta Inf.*, (15):281–302, 1981.

[12] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Softw. Eng.*, 4(SE-7):417–426, July 1981.

[13] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, (6):1976, 1976.

[14] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46, Polytechnic Institute of Brooklyn Press, New York, 1971.

[15] C. Seitz. *System Timing*, chapter 7. Addison-Wesley, 1980.

[16] U. Weiser and A. Davis. *A Wavefront Notation Tool for VLSI Array Design.* Computer Science Press, 1981.

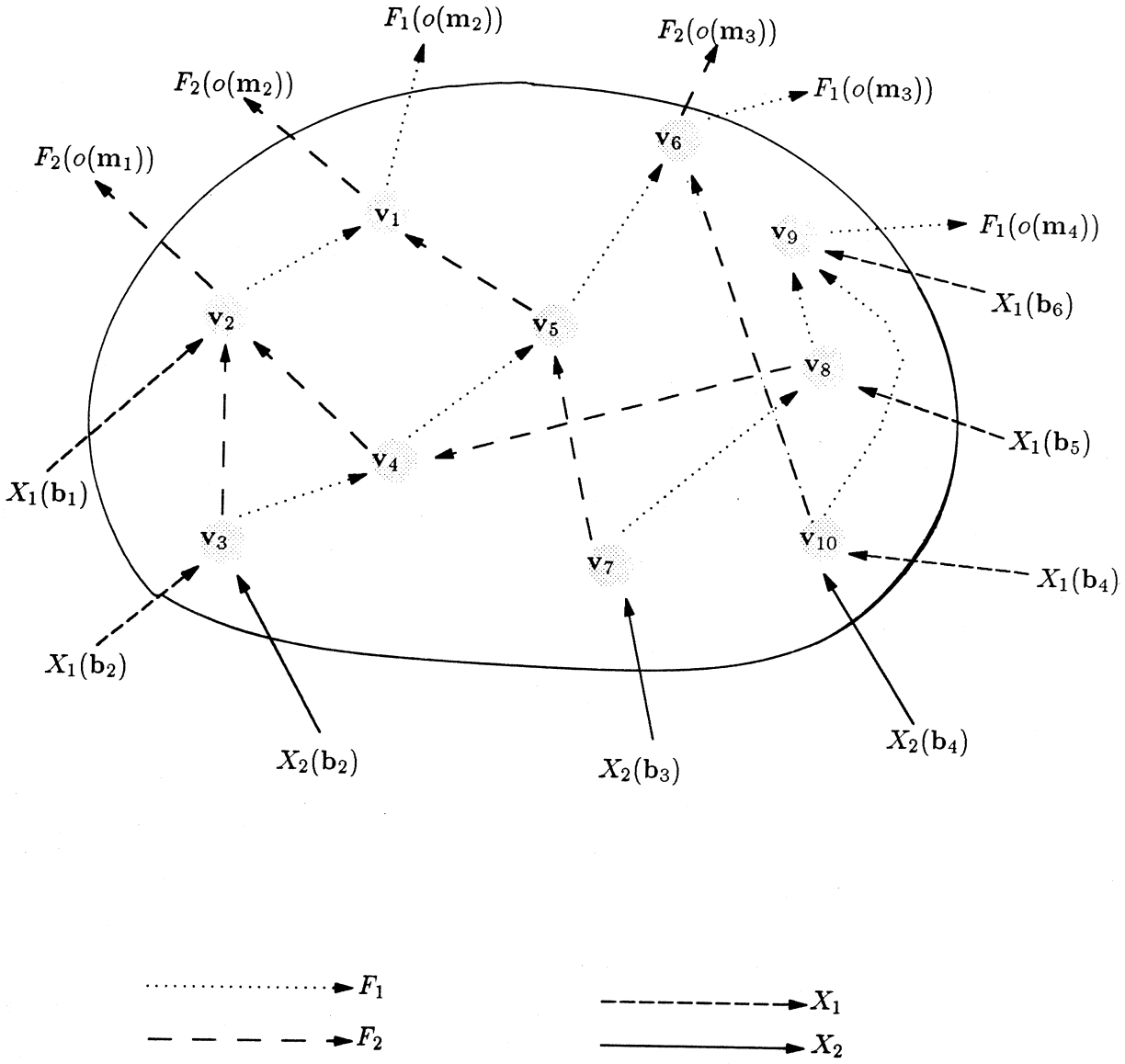

Figure 1: A **Crystal** program consisting of processes, communications between processes, data streams, external inputs, input mapping function ι which maps $\mathbf{v}_2 \mapsto \mathbf{b}_1$, $\mathbf{v}_3 \mapsto \mathbf{b}_2$, $\mathbf{v}_7 \mapsto \mathbf{b}_3$, $\mathbf{v}_{10} \mapsto \mathbf{b}_4$, $\mathbf{v}_8 \mapsto \mathbf{b}_5$, $\mathbf{v}_9 \mapsto \mathbf{b}_6$, external outputs, and output mapping function o which maps $\mathbf{m}_1 \mapsto \mathbf{v}_2$, $\mathbf{m}_2 \mapsto \mathbf{v}_1$, $\mathbf{m}_3 \mapsto \mathbf{v}_6$, $\mathbf{m}_4 \mapsto \mathbf{v}_9$.

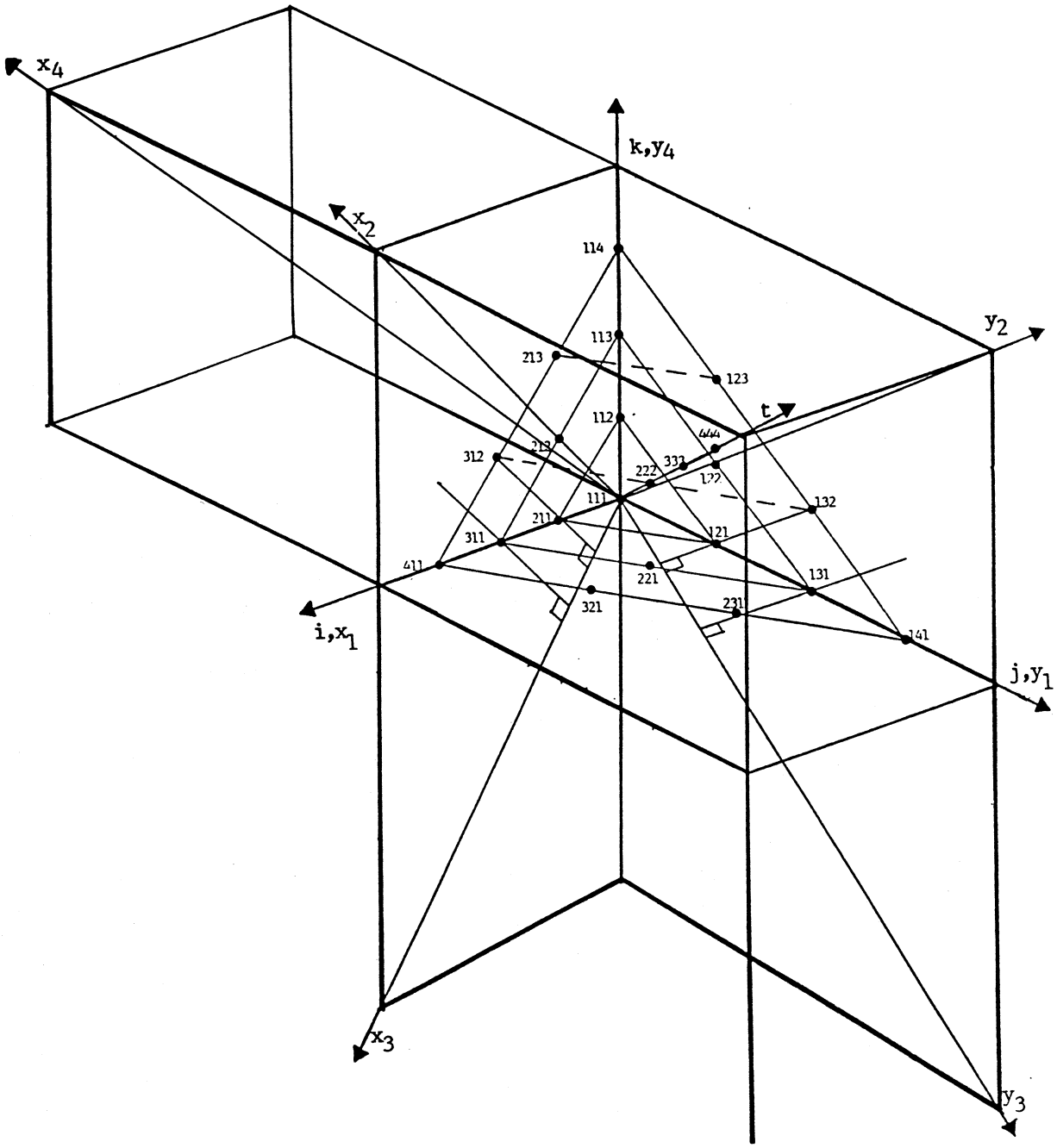

Figure 2: Process structure in the original coordinate system, and the new axis x_i, y_i, and t after space-time mapping $\mathbf{g}_i$ for $i = 1, 2, 3, 4$.

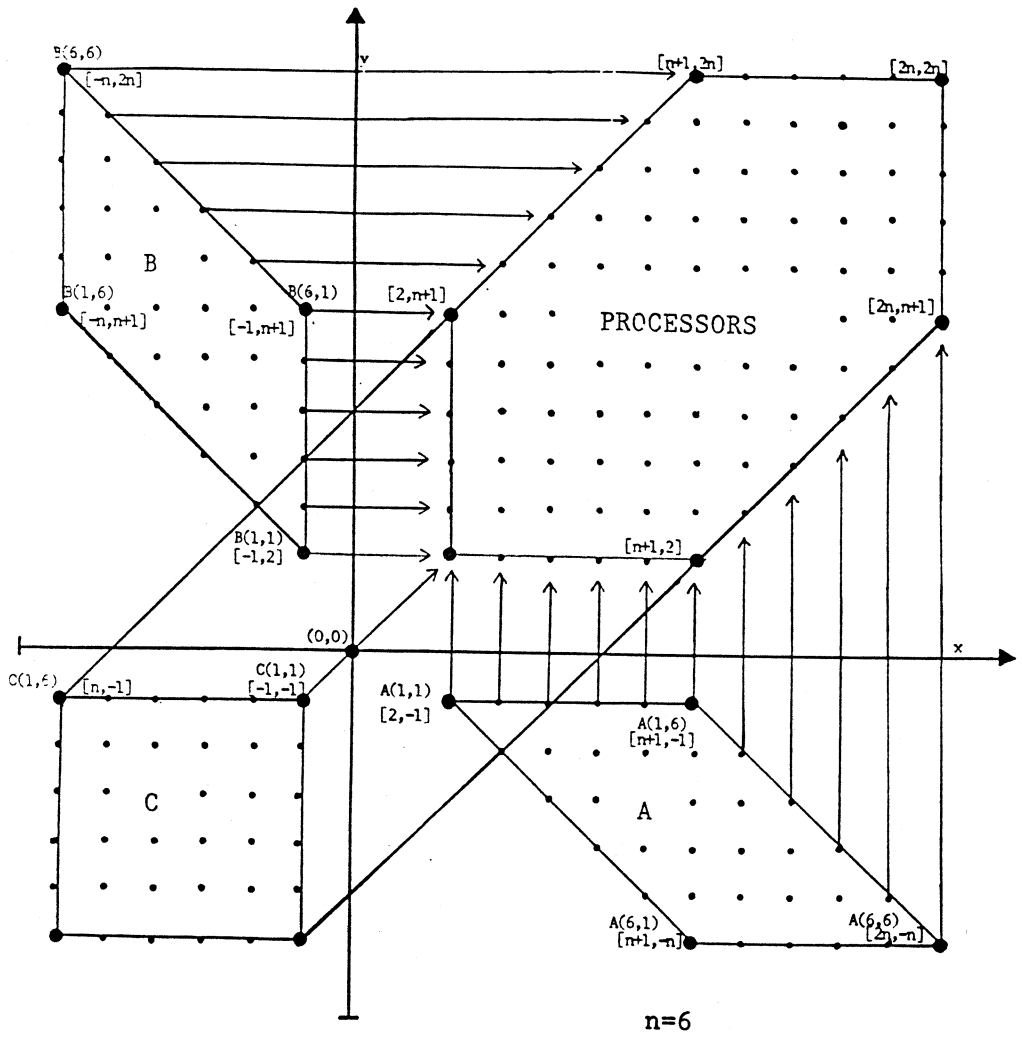

Figure 3: The systolic network and inputs using space-time mapping $\mathbf{g}_2$.

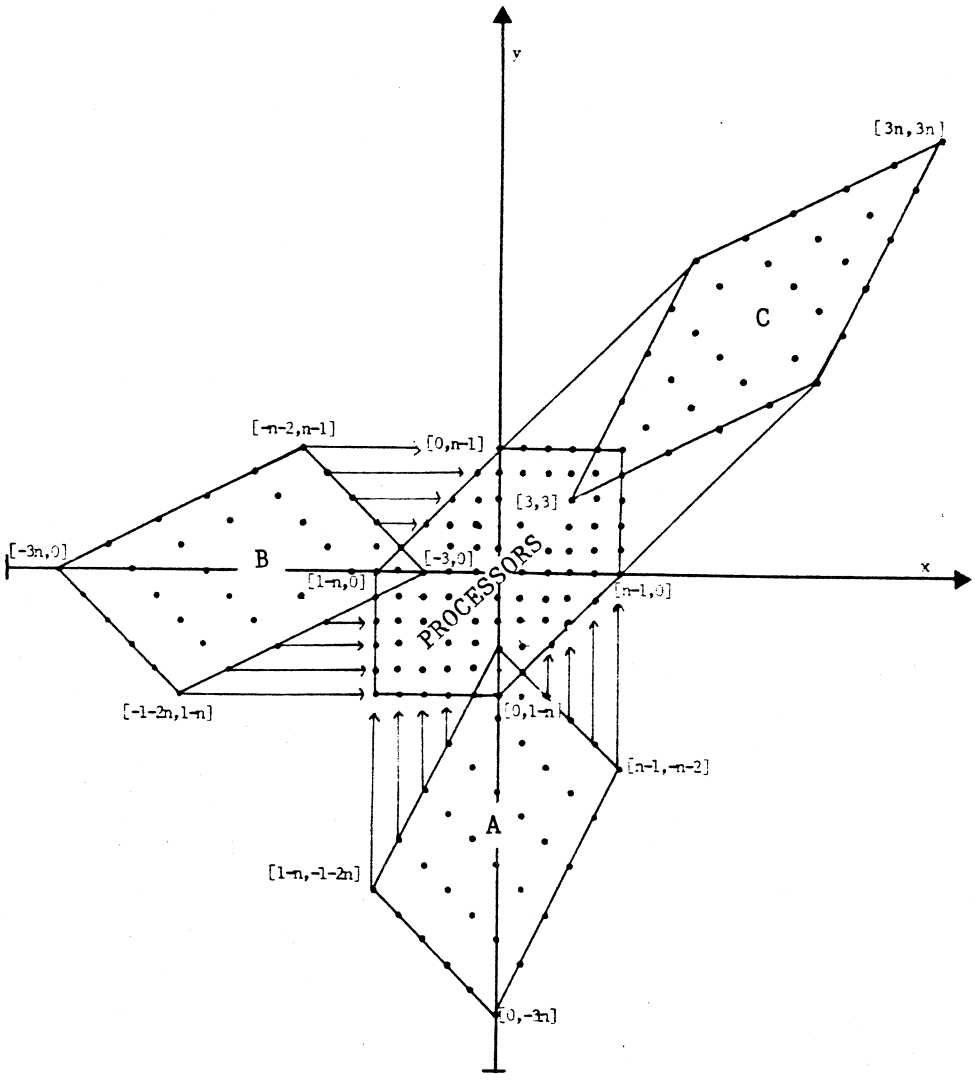

Figure 4: The systolic network and inputs using space-time mapping $\mathbf{g}_3$.