A programming language **Crystal** for describing parallel computations and for developing parallel programs is presented. A source program in **Crystal** is simply a straight-forward and highly readable mathematical definition. Concepts of process structure, wavefront sequence, process synchronization, and optimal timing for developing efficient parallel programs are introduced. Transformation rules for increasing the efficiency of a parallel program are presented. A synthesis method for incorporating pipelining into programs is presented. The synthesis of an efficient program from Warshall's transitive closure algorithm is illustrated.

To appear in the Proceedings of the IFIP's 86, September, 1986, Dublin, Ireland

Transformations of Parallel Programs in Crystal

Marina C. Chen

Research Report YALEU/DCS/RR-469
April 1986

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.

1 Introduction

A major software challenge is presented by the task of programming parallel machines effectively. One of the difficulties in software for parallel systems lies in the complex behavior arising from the interaction of a large scale of autonomous parallel processes. Unlike the computation described by a sequential program where it can be modeled as a sequence of state changes, the state changes in a parallel machine are concurrent and distributed, which makes the task of reasoning about a parallel program more difficult.

Suppose the computation going on in a collection of parallel processors can be described as the following: a set of *source* processors are initially started, each of which produces outputs which are sent to other processors, and cause those receiving processors to start execution, change state, and produce outputs, which in turn cause more processors to start computing and communicating. Critical to the description of such a computing system is a clear and unambiguous specification (deterministic or non-deterministic) of the *intended* state transition of a processor after which an output should be sent, and conversely, the intended state transition of a processor in which a particular received input should be used.

The programming notation *Communicating Sequential Processes* (CSP) pioneered by Hoare [6] provides guarded communication commands which serve this purpose. The central idea of CSP, embodied by its explicit communication commands, has been incorporated in the programming languages Occam [8], and widely adopted in languages, such as the extended C and Fortran, being supported on various parallel machines. Undoubtedly, such languages serve the purpose of unambiguously describing desired computations very well and are flexible enough for programmers to specify any desirable parallelism.

In the aspect of programming methodology, CSP is augmented with traces (expressed in a LISP-like language) and a set of algebraic laws on traces[7], which provide a framework in which the behavior of parallel processes can be verified. The mathematical treatment of CSP is quite elegant. However, one minor disturbing fact is that the language for programming and the language for developing programs are quite different. One of the consequences of the bilingual system is that program transformation is somewhat indirect, and must be guided by transformation on the traces which are in a different language.

A more serious question is whether CSP supports parallel programming at a suitable level? In a CSP program, often quite a few minute state changes and communications in each individual process in isolation are seen at once by a programmer. These isolated minute details seem both to obstruct and fail to provide a lucid global view of the parallel system. Consequently, the reasoning must be performed via composition of traces of interacting sequential processes, in a different language and outside of the program.

A more fundamental reason for this phenomenon lies in the underlying model in which a concurrent computation is modeled as a composition of sequential processes, where each sequential process is modeled by a sequence of events. This kind of 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 — being a logical unit. In fact, some of the most interesting parallel computations have as a logical unit a sequence of events "moving" in both space and time. In such case, CSP fails to capture these logically relevant events as a unit; rather, they are spread out to many sequential processes. On the other hand, what occurs in a single CSP sequential process turns out to be a sequence of independent, unrelated events.

Once we eradicate our prejudice against the new element in concurrent or parallel computations — the space component — we can then model an event purely as a mathematical object, because the very symmetry of space and time allows us to disregard them when programming. To develop an

implementation of a parallel program, a space-time mapping of events may ensue. Interestingly but perhaps not surprisingly, there are numerous ways of mapping events to space-time, each of which would result in a different CSP program. Thus the programming effort is elevated to a level that is beyond a particular space-time relationship of the sequential processes. This symmetrical view of space and time underlies the language **Crystal** [1].

Crystal is a high-level functional language consisting of formalized mathematical notation for describing large-scale parallel processes for various application domains. It provides, *inside* the language, constructs that encapsulate communications and minute state changes to form major state transitions of the entire assemblage; therefore a similar reasoning process for program correctness used in sequential programming may ensue. In **Crystal**, we don't need to reason about the behavior of processes as a sequence of events; instead we reason about them purely as mathematical objects. Proofs and programs are expressed in the same high-level mathematical notation. Last but not least, it is possible to derive *efficient* parallel implementations from a highly readable mathematical definition in a systematic way.

The remaining part of the paper is organized as follows: In Section 2, a program in the language **Crystal** is formally defined, and its semantics (fixed-point semantics [11], [9]) is given. **Crystal** programs written by users tend to be extremely straight-forward and highly readable. Often a problem definition is used directly as a source program. In Section 3, a few example programs are given. The simplicity of a **Crystal** program is supported by a synthesis methodology which, by successive stages of program transformations, allows the derivation of efficient parallel implementations. In Section 4, two vital concepts, process structures and wavefront sequences, for the synthesis and verification of efficient implementations are defined. In Section 5, two general rules for increasing the efficiency of parallel implementations are introduced as examples to illustrate program transformations. In Section 6, a program transformation that incorporates pipelining into a program is presented. In Section 7, the synthesis of an efficient program from Warshall's transitive closure algorithm [13] is illustrated.

2 A Crystal Program

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" ([11]) 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.

A **Crystal** program consists of a 9-tuple (program declaration),

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

and a system of recursion equations (program body)

$$\mathbf{F}(\mathbf{v}) = \phi_{\mathbf{v}}(\mathbf{F}\langle\tau^*_{\mathbf{V}}(\mathbf{v})\rangle, \mathbf{X} \circ \iota(\mathbf{v})), \ \forall \mathbf{v} \in P, \tag{1}$$

where

- P is a set of parallel *processes*, which is an index set.
- $\mathcal{D}$ 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.

- $\mathcal{F}$ is a set of identifiers for *data streams*, a $F \in \mathcal{F}$ also called a functional variable. Each functional variable $F_i \in \mathcal{F}$ ranges over a set of data streams, where a data stream is a function $E_i \stackrel{\text{def}}{=} [(P, \sqsubseteq) \rightarrow (D_i, \sqsubseteq)]$, the domain of continuous functions from $(P, \sqsubseteq)$ to $(D_i, \sqsubseteq)$.
- $\mathcal{X}$ is a set of *input variables*. Each input variable $X_i \in \mathcal{X}$ ranges over some *input functions* $[(B, \sqsubseteq) \rightarrow (D_i, \sqsubseteq)]$, where B is an index set, called the *input data structure* of program P.
- M, the output data structure, is also an index set.
- A *local processing function* $\phi_{\mathbf{v}}$ is a member of the family of functions $\{\phi\}_{\mathbf{v} \in P}$, and

$$\phi_{\mathbf{v}} : D_1 \times \cdots \times D_m \times D_{m+1} \times \cdots \times D_{m+l} \rightarrow D_1 \times \cdots \times D_m,$$

  for some non-negative integer m and non-negative integer l, and $\phi_{\mathbf{v}}(d_1, \ldots, d_m, d_{m+1}, \ldots, d_{m+l}) = (d'_1, d'_2 \ldots, d'_m)$, $d_i, d'_i \in D_i$.
- Function $\sigma_{\mathbf{v}}$ is a member of a family of functions $\{\sigma\}_{\mathbf{v} \in P}$, and each of its components $\sigma_{\mathbf{v},j}$, for $1 \leq j \leq n$ is

$$\sigma_{\mathbf{v},j} : D_1 \times \cdots \times D_m \times D_{m+1} \times \cdots \times D_{m+l} \rightarrow P,$$

  where $D_i \in \mathcal{D}$, for $1 \leq i \leq m + l$.
- The notation $\circ$ stands for functional composition.
- $\mathbf{X} \stackrel{\text{def}}{=} [X_1, \ldots, X_l]$ is a vector of input functions where $X_i \in [B \rightarrow D_{m+i}]$, $1 \leq i \leq l$.
- $\mathbf{F} \stackrel{\text{def}}{=} [F_1, \ldots, F_m]$ is a vector of identifiers for data streams (funcitonal variables) where $F_i \in \mathcal{F}$, $1 \leq i \leq m$
- $\mathbf{F}(\mathbf{v}) \stackrel{\text{def}}{=} [F_1(\mathbf{v}), \ldots, F_m(\mathbf{v})]$.
- 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):

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

  .
- A *communication function* $\tau_{\mathbf{v},j} : P \rightarrow P$, for $1 \leq j \leq n$, is defined as

$$\tau_{\mathbf{v},j} \stackrel{\text{def}}{=} \lambda \mathbf{u}.\sigma_{\mathbf{v},j}(\mathbf{F}(\mathbf{u}), \mathbf{X} \circ \iota(\mathbf{u}))$$

  where $F_i \in \mathcal{F}$ for $1 \leq i \leq m$ and $X_i \circ \iota(\mathbf{u}) \in D_{m+i}$ for $1 \leq i \leq l$.
- A composite communication function $\tau^*_{\mathbf{v},i}$, for $1 \leq i \leq m$, is defined by $\tau^*_{\mathbf{v},i} \stackrel{\text{def}}{=} \tau_{\mathbf{v},i_1} \circ \tau_{\mathbf{v},i_2} \circ \cdots \circ \tau_{\mathbf{v},i_{k_i-1}} \circ \tau_{\mathbf{v},i_{k_i}}$, where $i_j \in \{1, 2, \ldots, n\}$ for $1 \leq j \leq k_i$. The vector of composite communication functions $[\tau^*_{\mathbf{v},1}, \tau^*_{\mathbf{v},2}, \ldots, \tau^*_{\mathbf{v},m}]$ is denoted by $\tau^*_{\mathbf{v}}$.
- Function $\iota : P \rightarrow B$ is called an *input mapping function* which interfaces the input data structure and the process structure.
- Function $o : M \rightarrow P$ is called an *output mapping function* which interfaces the output data structure and the process structure.

- Functions $\sigma_{\mathbf{v}}$ 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,

$$\psi \stackrel{\text{def}}{=} \lambda \mathbf{F}.\lambda \mathbf{v}.\phi_{\mathbf{v}}(\mathbf{F}\langle \tau^*_{\mathbf{v}}(\mathbf{v})\rangle, \mathbf{X} \circ \iota(\mathbf{v}))$$

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 $\mathbf{F}^\infty$. The function $f_P \stackrel{\text{def}}{=} \lambda \mathbf{m}.\lambda \mathbf{X}.\epsilon \circ \mathbf{F}^\infty(o(\mathbf{m}))$ is said to be the function *implemented by* the parallel program P, where $\mathbf{m} \in M \Rightarrow o(\mathbf{m}) \in P$, and ϵ a projection (or identity) from $D_1 \times D_2 \times \cdots \times D_m$.

Remark: (Structured Programming) A **Crystal** program may contain a nested levels of **Crystal** programs since any function used in the definition of a program P, such as $\phi_{\mathbf{v}}$, can be implemented by another parallel program P'.

Definition 2.3 (Program Equivalence) Two programs P and P' are *equivalent* if and only if $f_P = f_{P'}$, 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

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

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

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

the communication function is

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

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 $\text{Ide} : (i,j) \mapsto (i,j)$ from $\mathcal{N}^2$ to $\mathcal{N}^2$, and the program body is

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

where $\tau^*_{(i,j)}(i,j) = \tau_{(i,j)}(i,j) = \sigma_{(i,j)}$. Since this system of recursion equations defines a pair of functions $[C,C]$ in which two components are identical, Equation (4) is simplified to Equation (3). 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 $\mathcal{N}$.

Program 2 (Ackerman)

$$A(x,y) = \begin{cases} x = 0 \rightarrow y+1 \\ y = 0 \rightarrow A(x-1,1) \\ else \rightarrow A(x-1, A(x,y-1)) \end{cases} \tag{5}$$

This recursive definition of the Ackerman's function also can be viewed as a parallel program:

$$P_{acker} = (\mathcal{N}^2, \{\mathcal{N}\}, \{A\}, \{\mathrm{Ide}\}, \mathcal{N}^2, \{\phi\}_{(x,y)\in\mathcal{N}^2}, \{\sigma\}_{(x,y)\in\mathcal{N}^2}, \iota, \mathrm{Ide}),$$

where $\mathcal{N}$ is the set of natural numbers; A is the functional variable; input constant function Ide as defined above; the input mapping function $\iota(i,j) \stackrel{\mathrm{def}}{=} \epsilon_2(i,j) + 1$ where ϵ_2 is defined above;

$$\phi_{(x,y),l}(d_1,d_2,d_3) = \begin{cases} x = 0 \rightarrow d_3 \\ y = 0 \rightarrow d_1 \\ else \rightarrow d_2 \end{cases} \qquad \text{for } l = 1,2,$$

and $\phi_{(x,y)} = [\phi_{(x,y),1}, \phi_{(x,y),2}]$; $\tau^*_{(x,y)} = [\tau^*_{(x,y),1}, \tau^*_{(x,y),2}]$, where

$$\tau^*_{(x,y),1}(x,y) = \tau_{(x,y),1}(x,y) = \sigma_{(x,y),1} = (x-1,1),$$
$$\sigma_{(x,y),2}(d) = (x-1,d)$$
$$\tau_{(x,y),2}(x',y') = \sigma_{(x,y),2}(A(x',y')),$$
$$\tau^*_{(x,y),2} = \tau_{(x,y),2} \circ \tau_{(x,y),3};$$

where
$$\tau^*_{(x,y),3}(x,y) = \tau_{(x,y),3}(x,y) = \sigma_{(x,y),3} = (x,y-1),$$

and finally, the program body

$$[A,A](x,y) = \phi_{(x,y)}([A,A]\langle\tau^*_{(x,y)}(x,y)\rangle, \mathrm{Ide} \circ \iota(x,y)), \quad \forall (x,y) \in \mathcal{N}^2. \tag{6}$$

The function $f_{P_{acker}} = \lambda(x,y).\epsilon_1 \circ [A,A]^\infty(\mathrm{Ide}(x,y))$ is implemented by P_{acker}, where $\epsilon_1 : (d_1,d_2) \mapsto d_1$ is a projection from $\mathcal{N}^2$ to $\mathcal{N}$.

Program 3 (Warshall)

$$C(i,j,k) = \begin{cases} k = 0 \rightarrow r(i,j) \\ k > 0 \rightarrow C(i,j,k-1) \vee (C(i,k,k-1) \wedge C(k,j,k-1)) \end{cases} \tag{7}$$

Warshall's algorithm for computing the transitive closure $C(i,j,n)$ of a binary relation $r(i,j)$ on a set of n elements $\{1,2,\ldots,n\}$ can be interpreted as a parallel program:

$$P_{warshall} = (P, \{Bool\}, \{C\}, \{r\}, \mathcal{N}^2, \{\phi\}_{(i,j,k)\in P}, \{\sigma\}_{(i,j,k)\in P}, \epsilon_{1,2}, \pi_n),$$

where $\mathcal{N}$ is the set of positive integers $\{1, 2, \ldots, n\}$;

$$P \stackrel{\text{def}}{=} (\mathcal{N} \times \mathcal{N} \times \{0\}) + \mathcal{N}^3,$$

$$Bool = \{true, false\},$$

$$\phi_{(i,j,k),l}(d_1, d_2, d_3, d_4) \stackrel{\text{def}}{=} \begin{cases} k = 0 \rightarrow d_4 \\ k > 0 \rightarrow d_1 \vee (d_2 \wedge d_3), \end{cases} \qquad l = 1, 2, 3;$$

and $\phi_{(i,j,k)} \stackrel{\text{def}}{=} [\phi_{(i,j,k),1}, \phi_{(i,j,k),2}, \phi_{(i,j,k),3}]$;

$$\tau^*_{(i,j,k)} \stackrel{\text{def}}{=} [\tau^*_{(i,j,k),1}, \tau^*_{(i,j,k),2}, \tau^*_{(i,j,k),3}]$$

where

$$\begin{aligned} \tau^*_{(i,j,k),1}(i,j,k) &= \tau_{(i,j,k),1}(i,j,k) = \sigma_{(i,j,k),1} = (i,j,k-1), \\ \tau^*_{(i,j,k),2}(i,j,k) &= \tau_{(i,j,k),2}(i,j,k) = \sigma_{(i,j,k),2} = (i,k,k-1), \\ \tau^*_{(i,j,k),3}(i,j,k) &= \tau_{(i,j,k),3}(i,j,k) = \sigma_{(i,j,k),3} = (k,j,k-1); \end{aligned}$$

input mapping function $\epsilon_{1,2}(i,j,k) = (i,j)$; and the output mapping function $\pi_n(i,j) = (i,j,n)$. The program body

$$[C,C,C](i,j,k) = \phi_{(i,j,k)}([C,C,C]\langle\tau^*_{(i,j,k)}(i,j,k)\rangle, r \circ \epsilon_{1,2}(i,j,k))$$

simplifies to Equation (7) because all three components are identical. Let $\epsilon_1 : (b_1, b_2, b_3) \mapsto b_1$ be a projection from $Bool^3$ to $Bool$. The function $f_{\mathcal{P}_{warshall}} = \lambda(i,j).\epsilon_1 \circ [C,C,C]^\infty(\pi_n(i,j))$ is implemented by $\mathcal{P}_{warshall}$.

Program 4 (Dynamic Programming)

$$C(i,j) = \begin{cases} j = i + 1 \rightarrow C_i \\ j > i + 1 \rightarrow \min_{i<k<j} h(C(i,k), C(k,j)) \end{cases} \tag{8}$$

The definition of dynamic programming can be posed in a general form where $C(i,j)$ is some cost function which is to be minimized. Therefore each instance of problems which can be solved by dynamic programming is a parallel program.

Program 5 (Matrix Multiplication)

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \times B(k,j) \text{ for } 0 \leq i,j < n, \tag{9}$$

It is the definition of multiplication of two matrices A and B. Readers may try to factor the above two mathematical definitions into the various parts of **Crystal** programs.

4 Process Structure and Wavefront Sequence

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 functional variables in the system. We also say that a process $\mathbf{v} \in P$ immediately depends on a constant ($\mathbf{c} \prec \mathbf{v}$) when some constant $\mathbf{c}$ 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 constants, not any other process $\mathbf{u} \in P$, then $\mathbf{v}$ is called a source process, or simply, a source. Let Sr denotes the set of source processes.

Definition 4.3 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 partially ordered set, and furthermore, there is no infinite decreasing chain from any process $\mathbf{u} \in P$, i.e., such that $(P \overset{*}{\prec})$ is a well-founded set.

Definition 4.5 (Data-independent Process Structure) If every $\sigma_{\mathbf{v},j}$ of program P is a nullary function for all $1 \leq j \leq n$, then $(P, \overset{*}{\prec})$ is called a *data-independent process structure*, otherwise, $(P, \overset{*}{\prec})$ is a data-dependent process structure.

Remark: Data-independent process structure is not necessarily static since it can be space-time variant. Strong results in synthesizing parallel programs which have data-independent process structure are possible [2,4,5]. There is no necessity to incur any run-time overhead for mapping parallel processes to processors for this class of programs since the problem 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.

Definition 4.7 (Dependency Vector) A *dependency vector* of a program that has a partially ordered vector space $(P, \overset{*}{\prec})$ as a process structure, 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 $\mathbf{v}$), where "$-$" is the inverse of the vector addition.

Definition 4.8 Let $(W, <)$ be a well-ordered set, and $\widehat{0}$ be its least element. We say that $n' \overset{!}{<} n$ (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 4.9 (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 $(W, <)$, (all sources belong to the first wavefront ($i = \widehat{0}$) of the sequence), and (ii) $w_n \overset{\text{def}}{=} \{\mathbf{v} : \forall \mathbf{u} \prec \mathbf{v}, \mathbf{u} \in w_k, k < n\}$, for $n > \widehat{0}$, (a process may belong to wavefront n if all of its dependent processes belong to wavefronts $k < n$). We use the notation $(w_i)_{i \in (W,<)}(P)$ to denote the sequence of wavefronts defined by a program P, and we call i the *wavefront expression* of the sequence.

Definition 4.10 (Optimal Wavefront) A wavefront sequence $(o_i)_{i \in (W,<)}(P)$ is optimal if for all $n > \widehat{0}$:

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

(a process belongs to the wavefront n if all of its dependent processes belong to wavefronts $k < n$ and there is at least one dependent process belonging to wavefront n' that is immediately less than wavefront n).

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. Furthermore, 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 some sequence of wavefronts of a program is quite easy, but ways of finding the optimal sequence of wavefronts may not be apparent. In the following, we illustrate how an optimal sequence can be obtained by an inductive procedure on some given wavefront sequence that is easy to obtain. Efficient parallel programs can then be derived based on the optimal sequence of wavefronts.

Remark: 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 [10].

5 Program Transformations

The rules of transformations introduced below are motivated by the constraints imposed by hardware The point stressed here is not so much the justification of the rules, but the uniformity of the syntax and semantics and the mathematical simplicity of program transformations in **Crystal**. Both of these rules are extremely useful and applies to a large number of problems [5], notably problems in solving systems of linear equations.

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

The time complexity of a **Crystal** program (at a particular level, where all processing functions and communication functions are assumed to cost a unit time step) 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, unfortunately, is 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). The time complexity therefore cannot be determined by the number of wavefronts alone; the time spent for each communication must also be taken into account. Let's 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 $O(n)$. Thus the elimination of large fan-in and fan-out degrees is essential in devising efficient parallel algorithms.

5.2 Locality of Communications

Locality of a communication is another factor that affects the amount of time it takes for the completion of the communication. Locality of communications is defined with respect to a given process structure P.

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 a process structure, 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 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 $\mathbf{c}$ appearing on the right hand side of a system of equation(s) while $\mathbf{u}$ appearing on its left-hand side.

Conversely,

Definition 5.5 The fan-out degree of a process $\mathbf{u}$ (or an input $\mathbf{c}$) is the total number of times a process $\mathbf{v}$ appearing on the left-hand side of a system of recursion equations while process $\mathbf{u}$ (or constant $\mathbf{c}$) 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 constant $\mathbf{c}$ in the case of fan-out degree) defined by the system.

5.4 Reducing the Fan-in Degrees

The transformation performed on a program is that of replacing an associative function that takes a large number of arguments by a program with a bounded fan-in degree that implements the function. As long as the replacing program implements the function correctly, the transformed program is equivalent to the original one.

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

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

such that

$$\begin{aligned} y &= z(v) \qquad \text{and} \\ z(l) &= \Phi(z, x, l, u, v, \oplus) \end{aligned} \tag{10}$$

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, where* $deg(x(l))$ *is the fan-in degree of* $x(l)$.

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.

5.5 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) *A set of equations* $\{F(l) = E_l(x) : l \in \mathcal{N}, u < l < v\}$, *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) An $(v-u-1)$-concurrent assignment $\{z(l) = x : u < l < v\}$ is equivalent to the sequence(s) of serial assignments defined by the recursion equation

$$z(l) = \Psi(z, x, l, u, v, w), \text{ where} \tag{11}$$

$$\Psi(z, x, l, u, v, w) \stackrel{\text{def}}{=} \begin{cases} l = w \rightarrow x \\ w < l < v \rightarrow z(l-1) \\ u < l < w \rightarrow z(l+1) \end{cases} \tag{12}$$

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

In Equation (12), 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 (11).■
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 a constant (an input), we let $w = u$, or symmetrically, $w = v$. When x is some value $F(l)$ depending on recursion variable l, let w be chosen so that $|w - l|$ is the minimum over all choices of w.

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 with the size in each dimension of $O(n)$ calling for $O(n^d)$ 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.

The method for incorporating pipelining into a program by space-time mapping is formulated for any program that has a process structure that is a partially ordered vector space. Space-time mapping allows the source program to be purely mathematical and void 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 also 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) For each program P, let an integer g_0 be an *initial synchronization point* and

$$\begin{aligned} & g(\mathbf{u}_1) + z_{\mathbf{v},1} \\ = \; & g(\mathbf{u}_2) + z_{\mathbf{v},2} \\ = \; & \cdots \\ = \; & g(\mathbf{u}_{k(\mathbf{v})}) + z_{\mathbf{v},k(\mathbf{v})}, \end{aligned} \tag{13}$$

be a set of $k(\mathbf{v}) - 1$ *synchronization equalities* (SE's) at process $\mathbf{v}$ for every $\mathbf{v} \in (P - Sr)$: where the process structure $(P, \overset{*}{\prec})$ is a partially ordered vector space, process $\mathbf{v}$ has $k(\mathbf{v})$ dependent processes $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{k(\mathbf{v})}$ $(\mathbf{u}_j \prec \mathbf{v})$, $z_{\mathbf{v},j}$, $1 \leq j \leq k(\mathbf{v})$ are $k(\mathbf{v})$ positive integers, $g(\mathbf{v}) = g_0$ for all $\mathbf{v} \in Sr$, and g is an integer-valued function.

Definition 6.2 (Optimal timing) For a given program P, if there exists an integer-valued function g over P that satisfies the set of synchronization equalities (13) at $\mathbf{v}$ with $k(\mathbf{v})$ positive integers $z_{\mathbf{v},j}$, where $1 \leq j \leq k(\mathbf{v})$, for every process $\mathbf{v} \in P$ of program P, then g is called a timing function of P. If furthermore, g is minimized over all timing functions 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 $\varphi : i \in W \mapsto t \in T$, where t is an integer and $i \overset{!}{<} i' \Rightarrow \varphi(i') = \varphi(i) + 1$. The function $g : P \to T$, defined by $g(\mathbf{v}) = \varphi(i)$ if $\mathbf{v} \in w_i$, is the optimal timing function of P.

Proof: Let $\hat{0}$ be the initial synchronization point, where $\hat{0}$ is the least element of $(W, <)$. For all $\mathbf{v} \in P - Sr$, let $z_{\mathbf{v},j} \overset{\text{def}}{=} g(\mathbf{v}) - g(\mathbf{u}_j)$ for each $\mathbf{u}_j \prec \mathbf{u}$. Since $g(\mathbf{v}) - g(\mathbf{u}_j) > 0$ by the definition of wavefront sequence and that of function φ, $z_{\mathbf{v},j}$ is a positive integer. 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 on the contrary, there exists some timing function g' such that $g'(\mathbf{v}) < g(\mathbf{v})$ for some $\mathbf{v} \in P - Sr$. Let $S \overset{\text{def}}{=} \{\mathbf{v} : g'(\mathbf{v}) < g(\mathbf{v}), \mathbf{v} \in (P - Sr)\}$, and choose a least element $\mathbf{v} \in (S, \overset{*}{\prec})$, i.e., for all $\mathbf{u} \in S$, either $\mathbf{v}$ and $\mathbf{u}$ are not related or $\mathbf{v} \overset{*}{\prec} \mathbf{u}$. 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 funciton and satisfies the SE's at $\mathbf{v}$, there exist a positive integer z, $g'(\mathbf{v}) = g'(\mathbf{u}_j) + z$. Now $z = g'(\mathbf{v}) - g'(\mathbf{u}_j) < g(\mathbf{v}) - g(\mathbf{u}_j) = 1$ implies that z is not positive, 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 $W \overset{\text{def}}{=} \{g(\mathbf{v}) : \mathbf{v} \in P\}$.

Proof: Since (1) $g(\mathbf{v})$ is integer-valued, (2) $g(\mathbf{v}) = g_0 \leq g(\mathbf{u})$ for for any $\mathbf{v} \in Sr$ and for all $\mathbf{u} \in P$, $(W \overset{\text{def}}{=} \{g(\mathbf{v}) : \mathbf{v} \in P\}, <)$ is a well-ordered set 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, refering to the SE's it satisfies, then for every $\mathbf{v} \in P$, there exists some j, $1 \leq j \leq k(\mathbf{v})$ such that $z_{\mathbf{v},j} = 1$. Otherwise $z_{\mathbf{v},j}$ for all $1 \leq j \leq k(\mathbf{v})$ 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. We have $\mathbf{v} \in o_i$ if $i = g(\mathbf{v})$, and there exists $\mathbf{u}_j \in o_{i-1}$ because $g(\mathbf{u}_j) = g(\mathbf{v}) - 1$. Since $i - 1 \overset{!}{<} i$, $(o_i)_{i \in W}$ indeed is the optimal wavefront sequence. ∎

Proposition 6.5 (Constructing optimal timing function) Let $(w_i)_{i\in(W,<)}$ be a wavefront sequence for program P. Then an optimal timing function g and the optimal wavefront can be constructed as follows:
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 $i \in W$, $i > \hat{0}$, 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 $1 \le j \le k(\mathbf{v})$ and $z_{\mathbf{v},j}$ is minimized.
   (b) assigning $g(\mathbf{v}) := g(\mathbf{u}_j) + z_{\mathbf{v},j}$, for some $j \in \{1, 2, \ldots, k(\mathbf{v})\}$.

Then g is the optimal timing function, and $(o_i)_{i\in(W,<)}$ 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: By induction on $(w_i)_{i\in(W,<)}$, g is the optimal timing function. By Theorem 6.4, $(o_i)_{i\in(W,<)}$ is the optimal wavefront sequence. ∎

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 an 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:

Definition 6.6 (Pipelining) A set of processes $R \stackrel{\text{def}}{=} \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$, $R \subseteq P$, of a program P is said to be *pipelined through* a given processor s if there exists a 1-1 function $h : \mathbf{v} \in R \mapsto t \in \mathcal{N}$ that is also a strictly monotonic function from $(R, \overset{*}{\prec})$ to $(\mathcal{N}, <)$, i.e., $h(\mathbf{u}) < h(\mathbf{v})$ if $\mathbf{u} \prec \mathbf{v}$. In this case, we say that the *pipelining factor* of the processor is n, and h is the pipelining function of processor s.

Definition 6.7 (Space-time mapping) A space-time mapping $\mathbf{g} \stackrel{\text{def}}{=} (g_s, g_t)$ of a program P is a pair of functions $g_s : \mathbf{v} \in P \mapsto s \in S$ and $g_t : \mathbf{v} \in P \mapsto t \in T$ such that all of the processes $\mathbf{v}$ in the set $R_s \stackrel{\text{def}}{=} \{\mathbf{u} : f_s(\mathbf{u}) = s\}$ for some s, pipeline through s with pipeline function $g_t|R_s$, the restriction of g_t to $R_s \subseteq P$.

Proposition 6.8 A space-time mapping $\mathbf{g}$ is a one-to-one function, and its inverse $\mathbf{g}^{-1}$ is well-defined.

Definition 6.9 An optimal space-time mapping $\mathbf{g} = (g_s, g_t)$ of a program P is a space-time mapping that g_t is the optimal timing function of P.

Remark: An optimal space-time mapping $\mathbf{g} = (g_s, g_t)$ of a program 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.

Definition 6.10 (Optimal pipelined program) Given a program

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

and its optimal space-time mapping $\mathbf{g}$, Another program

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

$$\hat{\mathbf{F}}(s,t) = \widehat{\phi}_{(s,t)}(\hat{\mathbf{F}}\langle\hat{\tau}^*_{(s,t)}(s,t)\rangle, \mathbf{X}\circ\hat{\iota}(s,t)),\ \forall (s,t)\in S\times T, \tag{14}$$

where

$$\begin{aligned}
&\hat{\mathbf{F}} \stackrel{\text{def}}{=} \mathbf{F}\circ\mathbf{g}^{-1}\\
&\widehat{\phi}_{(s,t)} = \lambda\mathbf{v}.\phi_{\mathbf{v}}(\mathbf{g}^{-1}(s,t))\\
&\hat{\sigma}_{(s,t),i} = \lambda\mathbf{v}.\sigma_{\mathbf{v},i}(\mathbf{g}^{-1}(s,t))\\
&\hat{\tau}_{(s,t),i} \stackrel{\text{def}}{=} \lambda(s',t').\hat{\sigma}_{(s,t),i}(\hat{\mathbf{F}}(s',t'), \mathbf{X}\circ\hat{\iota}(s',t'))\\
&\hat{\iota} \stackrel{\text{def}}{=} \iota\circ\mathbf{g}^{-1}\\
&\hat{o} \stackrel{\text{def}}{=} o\circ\mathbf{g}^{-1}
\end{aligned}$$

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

Proposition 6.11 Program $\hat{P}$ is equivalent to program P.

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

7 Program Synthesis from Warshall's Algorithm

The following is an example to illustrate the synthesis of an efficient parallel program from Warshall's algorithm $P_{warshall}$. First, large fan-in and fan-out degrees are eliminated, which results in an equivalent, hence correct program $P_{tran-clo}$. Next, a space-time mapping $\mathbf{g}$ is constructed for $P_{tran-clo}$ and an optimal pipelined program $\hat{P}_{tran-clo}$ with respect to $P_{tran-clo}$ by $\mathbf{g}$ is obtained.

7.1 Eliminating large fan-out degree

The fan-in degree of program $P_{warshall}$ is 3. Its fan-out degree is n because the fan-out degree of $C(i,k,k-1)$ and $C(k,j,k-1)$ are both n, which is unbounded and grows with the problem size. To eliminate the unbounded fan-out degree, the two n-concurrent assignments in Equation (7) shall

be replaced:

$\{a(i,k)(j) = C(i,k,k-1) : 0 < j < n+1\}$ such that
$C(i,j,k) = E(a(i,k)(j))$

$\{b(k,j)(i) = C(k,j,k-1) : 0 < i < n+1\}$ such that
$C(i,j,k) = E'(b(k,j)(i))$.

where

$a(i,k)$ is a function of j

$b(k,j)$ is a function of i

$$E \stackrel{\text{def}}{=} \lambda A. \begin{cases} k = 0 \rightarrow r(i,j) \\ k > 0 \rightarrow C(i,j,k-1) \vee (A \wedge C(k,j,k-1)) \end{cases}$$

$$E' \stackrel{\text{def}}{=} \lambda B. \begin{cases} k = 0 \rightarrow r(i,j) \\ k > 0 \rightarrow C(i,j,k-1) \vee (C(i,k,k-1) \wedge B) \end{cases}$$

Now applying function Ψ in Proposition 5.9 to appropriate arguments of Equation (7), two equations of the form of Equation (11) are obtained:

$$a(i,k)(j) = \Psi(a(i,k), C(i,k,k-1), j, 0, n+1, k), \qquad \text{and}$$
$$b(k,j)(i) = \Psi(b(k,j), C(k,j,k-1), i, 0, n+1, k).$$

Expanding the definition of function Ψ and redefining function $a(i,k)$ and $b(k,j)$ so that $a(i,k)(j)$ is written as $a(i,j,k)$ and $b(k,j)(i)$ is written as $b(i,j,k)$, we obtain a system of Equations

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

and

$$b(i,j,k) = \begin{cases} i = k \rightarrow C(k,j,k-1) \\ k < i < n+1 \rightarrow a(i-1,j,k) \\ 0 < i < k \rightarrow a(i+1,j,k) \end{cases} \tag{15}$$

such that

$$C(i,j,k) = \begin{cases} k = 0 \rightarrow r(i,j) \\ k > 0 \rightarrow C(i,j,k-1) \vee (a(i,j,k) \wedge b(i,j,k)) \end{cases}$$

Now the system of equations (15) is a **Crystal** program consisting of a system of first-order recursion equations with fan-in and fan-out degree three. The new program $P_{tran-clo} =$

$$(P, \{Bool\}, \{a,b,C\}, \{r\}, \mathcal{N}^2, \{\phi'\}_{(i,j,k)\in P}, \{\sigma'\}_{(i,j,k)\in P}, \epsilon_{12}, \pi_n),$$

is similar to the original program $P_{warshall}$ except that two more recursion variables are added and some sub-program definitions are added to implement the high fan-out degrees of $C(i,k,k-1)$ and $C(i,j,k-1)$.

7.2 Constructing a space-time mapping

Proposition 7.1 Define

$$\begin{cases} (i \geq k) \wedge (j \geq k) \rightarrow \\ z_{(i,j,k)}(0,0,1) = z_{(i,j,k)}(1,0,0) = z_{(i,j,k)}(0,1,0) = 1 \\ (i < k) \wedge (j \geq k) \rightarrow \\ z_{(i,j,k)}(0,0,1) = 3,\ z_{(i,j,k)}(0,1,0) = z_{(i,j,k)}(-1,0,0) = 1 \\ (i \geq k) \wedge (j < k) \rightarrow \\ z_{(i,j,k)}(0,0,1) = 3,\ z_{(i,j,k)}(1,0,0) = z_{(i,j,k)}(0,-1,0) = 1 \\ (i < k) \wedge (j < k) \rightarrow \\ z_{(i,j,k)}(0,0,1) = 5,\ z_{(i,j,k)}(-1,0,0) = z_{(i,j,k)}(0,-1,0) = 1, \end{cases} \tag{16}$$

in which whenever $i = k$, the term $z_{(i,j,k)}(1,0,0) = 1$ or the term $z_{(i,j,k)}(-1,0,0) = 1$ diminishes, and whenever $j = k$, the term $z_{(i,j,k)}(0,1,0) = 1$ or $z_{(i,j,k)}(0,-1,0) = 1$ diminishes. Let

$$g(i,j,k) = \begin{cases} k = 0 \rightarrow 2 \\ k > 0 \rightarrow 3k + |j-k| + |i-k|. \end{cases} \tag{17}$$

Then $g(i,j,k)$ is the optimal timing function of program $P_{tran-clo}$.

Proof: First we show that g is the optimal timing function by showing that it satisfies the SE's at every (i,j,k). The proof can be split into four cases:

1. $(i \geq k) \wedge (j \geq k)$,

$$\begin{aligned} & g(i,j,k) = \\ & g(i,j,k-1) + z_{(i,j,k)}(0,0,1) \\ & = 3(k-1) + (j-(k-1)) + (i-(k-1)) + 1 \\ & = g(i-1,j,k) + z_{(i,j,k)}(1,0,0) \\ & = 3k + (j-k) + ((i-1)-k) + 1 \\ & = g(i,j-1,k) + z_{(i,j,k)}(0,1,0) \\ & = 3k + ((j-1)-k) + (i-k) + 1 \\ & = i + j + k = 3k + |j-k| + |i-k| \end{aligned}$$

   Note that if $k = i$, the term $|(i-1)-k|$ in the equality diminish, therefore the term $|(i-1)-k|$ appears only if $i > k$ and it is always non-negative. Similarly for the case $k = j$.

2. $(i < k) \wedge (j \geq k)$,

$$\begin{aligned} & g(i,j,k) = \\ & g(i,j,k-1) + z_{(i,j,k)}(0,0,1) \\ & = 3(k-1) + (j-(k-1)) + ((k-1)-i) + 3 \\ & = g(i+1,j,k) + z_{(i,j,k)}(-1,0,0) \\ & = 3k + (j-k) + (k-(i+1)) + 1 \\ & = g(i,j-1,k) + z_{(i,j,k)}(0,1,0) \\ & = 3k + ((j-1)-k) + (k-i) + 1 \\ & = 3k - i + j = 3k + |j-k| + |i-k|, \end{aligned}$$

   where if $k = j$, the term $|(j-1)-k|$ diminishes.

3. $(i \geq k) \wedge (j < k)$: the proof is similar to case 2.

4. $(i < k) \wedge (j < k)$:

$$
\begin{aligned}
& g(i,j,k) = \\
& g(i,j,k-1) + z_{(i,j,k)}(0,0,1) \\
& = 3(k-1) + ((k-1)-j) + ((k-1)-i) + 5 \\
& = g(i+1,j,k) + z_{(i,j,k)}(-1,0,0) \\
& = 3k + (k-j) + (k-(i+1)) + 1 \\
& = g(i,j+1,k) + z_{(i,j,k)}(0,-1,0) \\
& = 3k + (k-(j+1)) + (k-i) + 1 \\
& = 5k - i - j = 3k + |j-k| + |i-k|
\end{aligned}
$$

It is optimal because for any (i,j,k), there exists some $\mathbf{d}$, $\mathbf{d} \in \{(0,1,0),(1,0,0),(0,-1,0),(-1,0,0)\}$, such that $z_{(i,j,k)}(\mathbf{d}) = 1$, which is the least possible value for $z_{(i,j,k)}$. ∎

Remark: It is an optimal timing function and can be obtained constructively as described in Proposition 6.5 by using the following wavefront sequence.

Proposition 7.2 For all $(i,j,k) \in P$, let

$$
l(i,j,k) = \begin{cases} k = 0 \rightarrow (0,0) \\ k > 0 \rightarrow (k, |j-k| + |i-k|), \end{cases}
$$

Let $\overset{2}{<}$ be the lexicographical ordering on the set of pairs $L \overset{\text{def}}{=} \{l(i,j,k) : (i,j,k) \in P\}$, then $(L, \overset{2}{<})$ is a well-ordered set. The sequence $(w_r)_{r \in (L, \overset{2}{<})}$ is a wavefront sequence for program $P_{tran-clo}$.

Proof: To see that l is a wavefront expression is quite straightforward. The first component k is determined by the third equation in System (15). With k fixed, the dependency in terms of i and j are clear from the top two equations. ∎

Proposition 7.3 Let g_s be defined as $g_s(i,j,k) = (i,j)$, g be the optimal timing function in Proposition 7.1. Then $\mathbf{g} \overset{\text{def}}{=} [g_s, g]$ is an optimal space-time mapping of program $P_{tran-clo}$.

Proof: Let $R_{(i,j)} \overset{\text{def}}{=} \{(i,j,k) : k \in \mathcal{N}\}$ 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 $\mathbf{g}$ is a space-time mapping. It is the optimal mapping because g is the optimal timing function. ∎

Having obtained the space-time mapping $\mathbf{g}$, program $P_{tran-clo}$ can now be transformed to $\hat{P}_{tran-clo}$ by applying Definition 6.10.

An efficient parallel program may require additional optimizations, such as elimination of "time-variant" tests of boolean predicates. Predicates such as $i = k$ and $j = k$ in $P_{tran-clo}$ which are "t-dependent" in $\hat{P}_{tran-clo}$. The reason for such elimination is to avoid the storage and operations on the indices, which is particularly critical in a VLSI implementation of a **Crystal** program. The computation of indices, which are integers, is quite expensive in this case in comparison with the Boolean-value computation of the transitive closure. The local tests of predicates in each processes can be replaced by additional data streams which do not contain time-variant boolean predicates. An illustration of such *control* optimization and others by program transformations can be found in [3]. Program $\hat{P}_{tran-clo}$ with control signal optimization incorporated becomes an implementation that is similar to the program appearing in reference [12].

8 Concluding Remarks

In this paper, it is demonstrated that conventional mathematical notation serves well as a programming language. In particular, it is very suitable for describing parallel programs. It is perhaps not a coincidence that a problem definition, viewed in the right light, suddenly becomes a parallel program, since it is never a mathematical concern that a definition ought to be executed on a sequential machine.

The semantics of a **Crystal** program is based on the classical fixed-point approach. Together with the abstraction and composition of functions, they make it possible to treat large complex problems just as if they were simple problems. They also allow implementation issues be resolved, resource utilizations be optimized, and insights about the problem be incorporated into a program by transformations that preserve correctness.

The high-level mathematical notation, with functions, sets and recursion equations, is one from which not only a suitable programming language is derived, but also a legacy of theories and techniques is inherited for its application in program synthesis. Perhaps because it is a suitable notation for parallel programming, it has also prompted many ideas which lead to the transformation rules and synthsis method. The repertoire of transformation rules, synthesis methods, and optimization techniques is bound to grow as we study more ways to compute by employing parallelism, as we have more insight into the problems to be solved, as we try to map programs to machines with various different network topologies, and as we have more experience in running programs on parallel machines. However, they can all be formulated in **Crystal**, and incorporated into programs by rigorous transformations.

References

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

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

[3] M. C. Chen. *A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI,* 13th ACM POPL Proceedings, page 131–139, January 1986.

[4] 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*, page , October 1985.

[5] M. C. Chen. *Crysal: A Synthesis Approach to Programming Parallel Machines,* To appear in the Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN, August 1985.

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

[7] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall, International series in Computer Science, 1985.

[8] INMOS Limited. *OCCAM Programming Manual.* Prentice Hall, International series in Computer Science, 1984.

[9] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[10] C. Seitz. System Timing, In *Introduction to VLSI Systems*, C. A. Mead and L. Conway, Chapter 7. Addison-Wesley, 1980.

[11] D.S. Scott and C. Strachey *Toward a Mathematical Semantics for Computer Languages*, Proceedings of the Symposium on Computers and Automata. Polytechnic Institute of Brooklyn Press, New York, 1971.

[12] Jan L. A. van de Snepscheut. *A Derivation of a Distributed Implementation of Warshall's Algorithm*, Technical Report CS8505 University of Groningen, The Netherlands, 1985.

[13] Stephen Warshall. *A Theorem on Boolean Matrices* Journal of ACM, 11-12, 1962.