To appear in the Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN Aug. 26-28, 1985

Crystal: A Synthesis Aproach to Programming Parallel Machines

Marina C. Chen

Research Report YALEU/DCS/RR-465
March 1986

Work supported by Office of Naval Research Contract N00014-85-K-0030 and National Science Foundation Grant DCR-8106181.

Abstract

This paper addresses two software issues in parallel processing: First, how to program parallel machines effectively, and second, how to realize the high performance promised by the parallel hardware technology. An overview of the language **Crystal** and a programming methodology for a class of computational problems are presented. **Crystal** provides high-level abstractions for the behavior of an ensemble of parallel processors, and is devised for the purpose of managing effectively the complexity of programming large scale parallel systems. To address the issue of effective utilization of hardware technologies, a design methodology is developed for synthesizing efficient systolic algorithms and architectures from problem definitions.

1 Introduction

Parallel processing is concerned with the effective use of hardware technologies that is inherently highly parallel. In order to expoit fully its potential, 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.

One of the difficulties of achieving efficient and reliable parallel software lies in the complex behavior arising from the interaction of a large scale of autonomous parallel processes. Beyond the sheer number of these processes, there is the complexity attributed to the concurrent flow of control and data distributed over the large number of processes.

Hardware technologies, on the other hand, constrain the way in which parallel computation can be efficiently organized. Dominant costs at the technological level often have profound implications in the designs at algorithmic and architectural levels. In order to achieve an efficient program design, one must take advantage of what the technology can offer, and minimize the costs associated with the constraints it imposes upon a design.

To manage effectively such complexity in programming and to realize the efficiency promised by the harware technology, we must strive for the advancements of software technologies in many areas, in particular, the devising of programming languages which provide high-level abstraction of the behavior of parallel processors, and programming methodologies for effectively partitioning and distributing tasks to a large-scale assemblage of processors.

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.

This paper presents an overview of the language **Crystal** and a programming methodology for a class of problems. In Section 2, language issues and programming methodologies are discussed. In Section 3, several example programs written in the language **Crystal** are given. In Section 4, we illustrate how to interpret a naive **Crystal** program so as to reveal its large scale parallelism. In Section 5, we describe the synthesis rules which allow the derivation of efficient parallel programs. In Section 6, we illustrate automatic incorporation of pipelining into programs. A resulting program has improved efficiency and embodies a systolic computation.

2 Programming Large Scale Parallel Machines

Programming a parallel machine is much more difficult than programming a sequential machine because unlike the computation in a sequential machine, where it can be modeled as a sequence of state changes and thus be reasoned with relative ease, the state changes in a parallel machine are concurrent and distributed, and a reasoning tool is not as easily attainable.

One may envision the computation going on in a collection of parallel processors being started by a set of *source* processors, 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 outputshould 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 [10] 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 [15], and widely adopted in languages supported by manufacturers of parallel machines, such as the extended C and Fortran languages on the Intel iPSC machines.

Undoubtably, such languages serve the purpose of unambiguously describing desired computations very well and are flexible enough for programmers to specify any desirable parallelism. But by merely having these parallel languages at hand, one does not come closer to resolving the problem of programming difficulty: i.e., how to reason about a parallel program which embodies concurrent and distributed state changes in a large-scale assemblage of processors.

2.1 Parallel Programming at a Suitable Level

In programming sequential machines, it is generally accepted that assembly languages are more flexible and may allow more efficient implementations than high-level languages, but are not suitable for large scale programming due to the burden of details they lay on programmers. Advancement in optimizing compilers of high-level languages eventually makes high-level programming both practical as well as desirable.

An analogous situation occurs in the choice of the level of descriptions in parallel programming, except that an even sharper contrast might be exhibited in the complexity of programming activity for the different choices made. Does the CSP style of language support the programming of parallel machines at a suitable level?

In a given CSP program, one can see very clearly the computation of each individual processor and, locally, how it explicitly communicates with others. However, when the number of processors become slightly larger and the interactions slightly more complicated, one quickly finds that it is very difficult to see through a CSP program — consisting of statements describing each minute state change in each individual process, and communication commands describing each interaction with other processes, — and still manage to obtain quickly a global

view of the parallel computation the program embodies. Often it becomes necessary to simulate the program in one's mind to see how processors are structured. The problem is that too many minute state changes and communications are seen at once by a programmer, and they obstruct a lucid global view of the parallel system. In other words, there is too much detail at too low a level. What is needed are language constructs that encapsulate explicit communications and minute state changes to form major state transitions of the entire assemblage, so that a similar reasoning process used in sequential programming may ensue.

Any sequential language augmented by a set of communication commands serves well as a parallel language, but only at the object language level in which a compiler generates code, and not at the source language level in which a programmer writes code. For instance, we use the high-level parallel language **Crystal**, and build compilers which generate object code in a CSP-style of language. A **Crystal** compiler currently targeted for Intel's iPSC hypercube multiprocessor uses the programming language C augmented with iPSC's communication commands as an object language.

2.2 Programming with High-level Functions

The language **Crystal** allows a functional style of programming in recursion equations, which aims at making the reasoning process of a parallel program easier. Abstractions of parallel operations into high-level functions are introduced.

What appears as a sequence of send and receive commands intermixed with local communications on a multiple number of processes in a CSP-style program, appears, in contrast, in a **Crystal** program as a single high-level functional call in which all actions said above are encapsulated. Moreover, the encapsulating functions can be defined at an abstract level which is independent of the *process structure* to be employed by a program, or the network topology of the machine on which the program is to be executed, thus eliminating unnecessary details when reasoning about a program.

Taking graph algorithms as an example, a user program can be specified in terms of the vertices and edges of a graph, rather than in terms of a particular data structure. A single functional call is able to describe operations such as "at each vertex, find the edge with minimum weight over all incident edge on that vertex","for each tree in a graph, find an edge with minimum weight that connects the tree with some other tree", etc. Such functions encapsulate multiple communications and multiple steps of local computations in a multiple number of processes.

To gain efficiency during execution, compilers make appropriate decisions on the process structures and routing schemes for a target network topology. For instance, the above-mentioned functions are compiled to employ the structure of an adjacency list when applied to graphs with the number of edges linearly proportional to the number of vertices. In the case of dense graphs, the structure of an adjacency matrix is used. Depending on the network topology of the multiprocessor used, each version of these functions is compiled to object code that uses particular embeddings of processes to processors and particular routing schemes suitable for the network.

With high-level functions as described above, a typical graph algorithm, such as finding the minimum weight spanning tree, can be described by several functional definitions. In Section 5.6, two high-level functions are defined and used in synthesizing systolic algorithms.

3 The Language Crystal

Crystal is a general purpose language for high level parallel programming. Syntactically, it consists mainly of formalized mathematical notations. Semantically, it is quite unlike any of

the imperative programming languages such as FORTRAN, PASCAL, or C, and is similar to functional languages such as LISP. However, its model of interpretation is data-driven, and therefore departs significantly from the demand-driven model of conventional applicative or functional languages.

The language **Crystal** consists of:

- *sets* for data types;
- *functions* for abstraction, where a function may be implemented either as sequential code or by an assemblage of parallel processes;
- *recursion equations*, which are the main constructs for parallelism;
- *bounded range*, which is basically a for loop constuct;
- and *unbounded minimalization*, as in recursive function theory, which is essentially a while loop construct, and allows the specification of recursions without an *a priori* bound.

With these constructs, **Crystal** is able to express homogeneous or inhomogeneous parallel systems of any network connectivity.

3.1 Definitions as Programs

At the top level, each parallel program in **Crystal** consists of a system of recursion equation(s). For example, the number of partitions of an integer k into integers less than or equal to m can be obtained by applying the following recursively defined function C to the pair of integers (m, k).

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

In Equation (1), we call i and j recursion variables and C a functional variable of the equation.

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.

$$C(i,j) = \begin{cases} j = i+1 \rightarrow C_i \text{ (the individual costs)}, \\ j > i+1 \rightarrow \min_{i<k<j} h(C(i,k), C(k,j)) \text{ (h is some function on the costs)}, \end{cases} \tag{2}$$

where recursion variables i and j range over integers $0 < i < j \leq n$, for some constant integer n. Another similar example,

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \times B(k,j) \text{ for } 0 \leq i,j < n, \tag{3}$$

is a specification for the matrix multiplication, with functional variable C for the resulting product and recursion variables i and j for, respectively, the row and column index of matrices A and B.

3.2 Symmetry of Recursion Variables

Comparing **Crystal**'s recursion equations with the notation of recurrences, they seem to be merely notational variances of each other. However, the significance is that **Crystal** intends to treat all recursion variables in a symmetrical manner. Taking the problem of LU decomposition

of matrices as another example: In **Crystal**, it is defined as the following system of recursion equations which is essentially the standard definition of LU decomposition. It describes that the decomposition of the matrix A is obtained iteratively as index k ranges from 0 to n, and at each iteration, a column of the L matrix and a row of the U matrix are computed.

$$
\begin{aligned}
a(i,j,k) &= \begin{cases} k = 0 \rightarrow A(i,j) \\ 0 < k \leq n \rightarrow a(i,j,k-1) + L(i,k) \times (-U(k,j)) \end{cases} \\
L(i,k) &= \begin{cases} 1 \leq i < k \rightarrow 0 \\ i = k \rightarrow 1 \\ k < i \rightarrow a(i,k,k-1) \times U(k,k)^{-1} \end{cases} \\
U(k,j) &= \begin{cases} 1 \leq j < k \rightarrow 0 \\ k \leq j \rightarrow a(k,j,k-1) \end{cases}
\end{aligned}
\tag{4}
$$

In the system of equations, we call i, j, and k recursion variables; they correspond to subscripts (e.g. indices of matrix elements) and superscripts (e.g. indices for iterations) in the notation of linear recurrences. In a recurrence, a subscript such as i or j is thought to be an index for a static location of some structured data, and a superscript such as k is thought to be an index for iterative computational steps. However, in the multi-dimensional design space of parallel programs, there are numerous different ways a subscript or superscript can be interpreted: statically as a location, dynamically as a computational step, or as a linear combination of both. For this reason, **Crystal** chooses to treat both subscripts and superscripts in recurrences in a symmetrical manner.

3.3 Expressive Power of Crystal

Formally, a **Crystal** program assumes the form of course-of-value recursion with unbounded minimalization, and thus it is general recursive. Practically, we have had some experience in expressing problems in various application domains in **Crystal**. We have found that problems in scientific computing and physics can be expressed very fluently, perhaps due to the mathematical nature of the language. On the side of symbolic computation, the kernal of a VLSI switch level simulator [1] has been described in **Crystal** as three levels of recursion equations [5]. Graph algorithms, such as finding connected components, minimum spanning trees, bi-connected components, etc. [11], can easily be expressed in **Crystal** as well. **Crystal** also serves as a hardware description language for various computer aided design tools. The descriptions of VLSI arithmetics, logic designs, and machine architectures [5], etc., in **Crystal**, serve as precise functional definitions of the hardware.

4 Parallel Interpretation of Crystal Programs

4.1 Process structure

Crystal interpretes a program based on a data-driven model. Take the straightforward definition given in Equation (3) for matrix multiplication as an example: each pair of indices (i,j) is interpreted as a *process* in the *process structure* $P \stackrel{\text{def}}{=} N^2$ which is a cartesian product of $N \stackrel{\text{def}}{=} \{1,2,\ldots,n\}$. The *input set* C of the program consists of the input data $C \stackrel{\text{def}}{=} \{A(i,j), B(i,j) : 1 \leq i,j \leq n\}$.

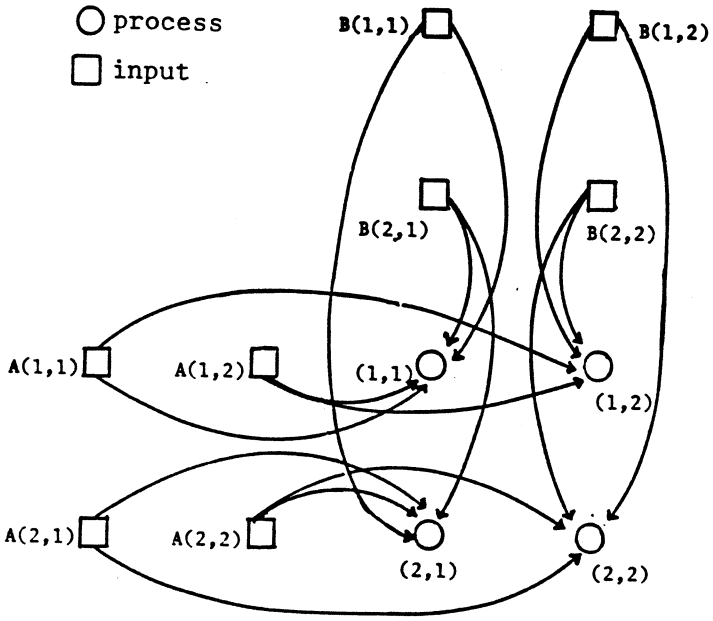

Figure 1: The DAG describing the data dependency of the multiplication of 2-by-2 matrices

4.2 Data Dependency Relation

The union of the process structure and the input set $P \cup C$ is ordered by a data dependency relation. We say that 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 right-hand side of a system of equations, and $\mathbf{u}$ appears on the left-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 $\mathbf{c} \in C$ ($\mathbf{c} \prec \mathbf{v}$) when some constant $\mathbf{c}$ appears on the right-hand side of the system while $\mathbf{v}$ appears on the left-hand side. A system (e.g. Equation 3) must be such that the transitive closure "$\overset{*}{\prec}$" (precedes) of "$\prec$" on all processes is a partial order, and that there is no infinite decreasing chain from any process $\mathbf{u} \in P$, i.e., the domain of process with relation "$\overset{*}{\prec}$", denoted by $(P, \overset{*}{\prec})$, is a well-founded set.

The set $(P \cup C, \overset{*}{\prec})$ can be depicted as a DAG (Directed Acyclic Graph), which is shown in Figure 1. It consists of nodes, where each node is either a process $\mathbf{u} \in P$ or some constant $\mathbf{c} \in C$, and directed edges, where an edge is emitting from node $\mathbf{u}$ to node $\mathbf{v}$ if $\mathbf{u} \prec \mathbf{v}$. Those nodes that have no incoming edges are called *sources*, thus all nodes in the input set are sources.

4.3 Data-driven Model and Execution Wavefront

Extracting a sequence of *execution wavefronts* from each **Crystal** program is useful in reasoning about the program and in synthesizing efficient parallel implementation from the program. The wavefront sequence of a program essentially captures the sequence of global state transitions in the assemblage of parallel processes, and thus provides a useful tool for reasoning about the program.

Our data-driven interpretation of a **Crystal** program envisions a parallel computation which starts at the sources, and is followed by other processes each of which starts execution when all

of its required inputs, or dependent data, become available. A sequence of *execution wavefronts* w_i for $i = 0, 1, 2, \ldots$, can be defined. All sources belong to the zero'th wavefront ($i = 0$) in the sequence, and a process belongs to the n'th wavefront if all of its dependent processes belong to wavefronts $n-1$ or less and there is at least one dependent process belonging to wavefront $n-1$, i.e.,

$$w_n \stackrel{\text{def}}{=} \{\mathbf{v} : \forall \mathbf{u} \prec \mathbf{v}, \mathbf{u} \in w_k, k < n \text{ and } \exists \mathbf{u}, \mathbf{u} \in w_{n-1}\}, \text{ for } n > 0.$$

For matrix multiplication, the sequence consists of only two wavefronts: w_0 containing all sources and w_1 containing all processes (i, j), $1 \leq i, j \leq n$.

4.4 Synchronous vs. Asynchronous Systems

The wavefront sequence of a program provides a reasoning tool independent of whether the parallel implementation of the program uses synchronous circuitry or an ensemble of asynchronous processors. Clearly, processes belonging to the same execution wavefront do not depend on one another. In an implementation of the parallel interpretation, such processes could be arranged to execute simultaneouly 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. 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+1}. Such is the case in a self-timed system [21].

4.5 Local Processing Functions

The local function each process should perform can be extracted from the definition. The local function performed may vary from process to process; hence, often a family of local functions, subscripted by the indices of the process, are defined. For the definition of matrix multiplication in Equation (3), the family of processing functions $G_{i,j}$ happens to be identical to a single function G which is defined as:

$$G(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots b_n) \stackrel{\text{def}}{=} \sum_{k=1}^{n} a_k \times b_k \tag{5}$$

The function G is defined with $2n$ formal parameters a_k and b_k for $k = 1, \ldots, n$.

4.6 Communication Functions

Similarly, the communications between processes can be extracted from the definition. Each formal parameter of a local processing function, at run time, might either be substituted by a functional value evaluated within the process, or by a value passed on by other processes. In the latter case, a communication function is defined for specifying which process is passing on a value for the parameter substitution. In the definition of matrix multiplication, all actual parameters are input constants, hence no communication functions are extracted. In the definition of integer partition 1, there are two different families of communication functions: $\tau_{1i,j} \stackrel{\text{def}}{=} (i-1, j)$ and $\tau_{2i,j} \stackrel{\text{def}}{=} (i, j-i)$. Note that in this example, the communication functions used at each process (i, j) depends only on the indices of the process. In general, a communication function may be data dependent.

To summarize, the local processing function of each process and the communications between processes can be extracted from a **Crystal** program which is written in a high-level functional notation in which explicit communication command is not necessary.

4.7 Complexity of the Naive Parallel Algorithm

Using one processor for each process in Equation (3), 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 above. Altogether $O(n^2)$ number of parallel processors are needed. The number of time steps needed is two, the total number of execution wavefronts. As exemplified by the definition of matrix multiplication, if a parallel program does not consider how computations and commuications are constrained by hardware, then the measure of its time complexity is too naive to be useful. In the next section, we show how to transform systematically a definition to a parallel program that does take into account realistic costs of hardware, and allows the wavefront number to serve as a valid measure of time complexity.

5 Efficiency of Parallel Programs

To address the issue of effective utilization of hardware technologies, we have developed a design methodology for synthesizing efficient systolic algorithms and architectures from problem definitions. Systolic algorithms, as exemplfied by a large body of literature [13], 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? Numerous reports [19,18], [17], [20], [14], [6], [7], [3], and [12] have appeared in the area of systhesis methodology for systolic algorithms. A detailed review on the relative strength of these different approaches and comparisons of the computational complexity of the mapping procedures appear in [7]. Mostly, the emphasis of these reported synthesis methods is on the mapping of a problem specification to a systolic architecture by a linear transform. The specification is mostly given in recurrences of one form or another. Very little attention has been paid to how to come up with a suitable problem specification that is amenable to a linear transformation in the first place. In [19] and [17], a problem specification is obtained from a FORTRAN loop by augmenting it with suitable indices. In [7], several guidelines (eg. first order recursion equations for nearest neighbor communications) for transforming specifications to a target specification are given, but no specific formal rules. In [12], the method first seeks linear transforms and then requires the user to modify the problem specification if the former fails. The process iterates until some linear transform is found, but no guideline as to how modification should proceed is given. Clearly, a unified methodology for synthesizing systolic algorithms, at the specification stage as well as at the mapping stage, is highly desirable.

With our synthesis methodology, a systolic algorithm can be generated systematically by a series of transformations on the description of problem definition, each aiming at reducing the dominant costs in the underlying hardware which might be incurred by the parallel implementation of the definition. A set of formal rules, and the necessary and sufficient conditions for the existence of linear designs are given. All specifications, from the initial problem definition to the target specification, and rules of transformations, are described in the same notation — the language **Crystal** [4]. This uniformity makes the tools that incorporate the methodology portable to any target machines.

5.1 Communication: An Implicit Cost

For each process in a given execution wavefront, it either communicates with other processes belonging to previous wavefronts, or receives a datum in the input set. The time it takes for a process to complete a communication, unfortunately, is not entirely independent of the number of destinations to which data must be sent (the fan-out degree), nor of the number of sources from which data must be received (the fan-in degree). The time complexity of a parallel algorithm, therefore, cannot be determined by the number of execution wavefronts alone; the time spent for each communication must also be taken into account.

Definition 5.1 *The amount of time for completing a single process (single source) to process (single destination) communication is called unit communication time.*

5.2 Locality of Communications

From the standpoint of hardware cost, the more local a communication is, the less the area and the shorter the time required for its implementation. *Locality* of a communication is another factor that affecting the amount time it takes for its completion. The concept of locality of communications is defined with respect to a given process structure P.

Definition 5.2 *If the process structure P of a program is an m-dimensional vector space over the rationals, then the path length between two processes $\mathbf{v} \stackrel{\text{def}}{=} (v_1, \ldots, v_m) \in P$ and $\mathbf{u} \stackrel{\text{def}}{=} (u_1, \ldots, u_m) \in P$ with respect to P is defined to be $|v_1 - u_1| + \cdots + |v_m - u_m|$, where each of the components u_i and v_i for $i = 1, 2, \ldots, m$ is an integer.*

Given the definition of path length with respect to a specific process structure, locality can be defined:

Definition 5.3 *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 less than a fixed constant.*

Definition 5.4 *The order of a system of recursion equations 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 $\mathbf{u} \prec \mathbf{v}$.*

5.3 Problem with Large Numbers of Fan-ins and Fan-outs

Due to the inherent physical constraints imposed by the driving capability of communication channels, power consumptions, heat dissipations, memory bandwidth, etc., data cannot be sent or received to or from a large number of destinations or sources in a unit communication time. Putting such constraints in algorithmic terms: The time it takes for a communication between two processes to complete depends on the fan-in and fan-out degrees, where the *fan-in degree* of a datum is defined as the number of data items it depends on, and *fan-out degree* is the number of data items dependent upon it. 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.

5.4 Counting Fan-in and Fan-out Degrees

The fan-in degree of a process $\mathbf{u}$ is obtained by counting the number of distinct processes $\mathbf{v}$ or constants $\mathbf{c}$ appearing on the right hand side of a definition. Conversely, the fan-out degree of a process $\mathbf{u}$ or a constant $\mathbf{c}$ is the number of times $\mathbf{u}$ (or $\mathbf{c}$) appears on the right-hand side while a process $\mathbf{v}$, each time with a different $\mathbf{v}$, appears on the left-hand side.

From Equation (3), the fan-in degree of each process (i,j) in the definition of matrix multiplication is $2n$ and the fan-out degree of any input data, $A(i,j)$ or $B(i,j)$, is n; both degrees are unbounded.

5.5 Bounded Fan-in and Fan-out degrees

Can the unbounded fan-in and fan-out degrees of a program be decreased so as to decrease its total time complexity? Is there any alternative cost one must pay for the bounded fan-in and fan-out degrees? One would expect that the number of execution wavefronts might increase as a result of decreased fan-in and fan-out degrees. It turns out that often the reduction in the number of unit communication times due to reduction in the fan-in and fan-out degrees exceeds the increase in the number of execution wavefronts. Thus the total time complexity is reduced.

5.6 Reduce Fan-in and Fan-out Degrees

The fan-in and fan-out degrees of a system of recursion equations can be reduced if the following high-level functions and rules of transformations are applied.

Definition 5.5 (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, 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.
Remark: Function Φ in Equation $z(l) = \Phi(z,x,l,u,v,\oplus)$ of Definition 5.5 encapsulates the computations and the communications of $v-u+1$ number of processes.

Definition 5.6 (concurrent assignment) *A set of equations $f(l) = H(x)$ for integer l, $u < l < v$, contains a set of concurrent assignments $\{z(l) = x : u < l < v\}$ if such a z exists and $f(l) = H(z(l))$.*

Proposition 5.7 (serial assignment) *A set of concurrent assignments $\{z(l) = x : u < l \leq v\}$ is equivalent to the sequence(s) of serial assignments defined by the recursion equation*

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

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

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

In Equation (7), fan-out degrees of x and $z(l)$ for $u \leq l < w$ and $w < l \leq v$ are all 1, and the fan-out degree of $z(w)$ equals 2 if $u < w < v$ and equals 1 if $w = u$ or $w = v$.

Remark: The choice of w in Proposition 5.7 concerns the issue of locality of the communication between value x and variable $z(w)$. When x is a constant, then $w = u$ or $w = v$. When x is some value $F(l)$ depending on recursion variable l, then w is chosen so that $|w - l|$ is the minimum over all choices of w.
Remark: Function Ψ in Proposition 5.7 encapsulates the communications of $v - u + 1$ number of processes.

5.7 An Example

In the matrix multiplication example, we note that Equation (3) contains an n-ary associative operation "$\sum$". The high fan-in degree is reduced by applying function Φ of Definition 5.5 to approprite arguments of Equation (3), and the high fan-out degrees of $A(i,k)$ and $B(k,j)$ are reduced by detecting concurrent assignments in the System and replace them with serial assignments. The steps of program transformations and some notations are given below.

Notation 5.8 *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 $\lambda x.E(x)$.*

First, Equation (3) is transformed to one that is in the form of Equation (6):

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

Expanding the definition of function Φ, and redefine $c(i,j)(k)$ as $c(i,j,k)$, we obtain

$$C(i,j) = c(i,j,n) \qquad \text{and}$$
$$c(i,j,k) = \begin{cases} k = 0 \rightarrow 0 \\ 0 < k \leq n \rightarrow c(i,j,k-1) + A(i,k) \times B(k,j) \end{cases} \tag{9}$$

Next, observe that Equation (9) contains $2n$ sets of concurrent assignments

$\{a(i,k)(j) = A(i,k) : 0 < j < n+1\}$ such that $c(i,j,k) = H_a(a(i,k)(j))$

$\{b(k,j)(i) = B(k,j) : 0 < i < n+1\}$ such that $c(i,j,k) = H_b(b(k,j)(j))$.

where

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

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

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

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

Now applying function Ψ in Proposition 5.7 to appropriate arguments of Equation (9), two equations of the form of Equation (7) are obtained:

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

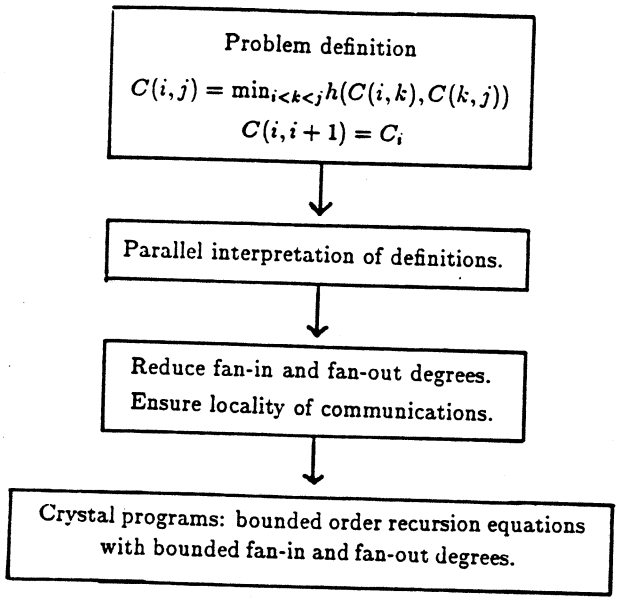

Figure 2: Program transformation.

Expanding the definition of function Ψ and redefine 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

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

and

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

such that

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

Remark: The order of the application of Definition 5.5 and Proposition 5.7 to Equation (3) can be interchanged.

Now the system of equations (10) is a **Crystal** program with fan-in and fan-out degree three. Figure 2 summarizes the steps of program transformations.

6 Incorporating Pipelining by Space-time Mapping

A naive implementation of System (10) derived above could use one processor for each process; however, after the execution of a process, a processor would be sitting idle, wasting resources. In general, a system of low-order recursion equations with bounded fan-in and fan-out degrees defined on a d-dimensional process structure 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.

6.1 Mapping Processes to Processors

System (10) derived above may be further improved by incorporating pipelining. From the stand-point of implementation, a process $\mathbf{v}$ in a system of recursion equations will be mapped to some physical functional unit, or processor s during execution, and once the process is terminated, another process can be mapped to the same processor. In fact, such re-use of the resource is the essence of *pipelining*. We call each execution of a process by a processor an *invocation* of the processor. Let t be an index for labeling the invocations so that the processes executed in the same processor can be differentiated, and let these invocations be labeled by strictly increasing non-negative integers. Then for a given implementation of a program, each process $\mathbf{v}$ has an alias $[s,t] \stackrel{\text{def}}{=} f(\mathbf{v})$, telling when (which invocation) and where (in which processor) it is executed. The key to an efficient parallel implementation of an algorithm is to find an appropriate one-to-one function f that maps a process $\mathbf{v}$ to its alias $[s,t]$ such that t will be non-negative and $t_2 > t_1$ if $\mathbf{v}_1 \prec \mathbf{v}_2$, where $[s_1,t_1] \stackrel{\text{def}}{=} f(\mathbf{v}_1)$ and $[s_2,t_2] \stackrel{\text{def}}{=} f(\mathbf{v}_2)$.

6.2 Data Dependency Vectors

To find such a mapping, we require that the domains of the recursion variables of a **Crystal** program be vector spaces, and that appropriate vector addition and scaler vector product be defined. Though each of the recursion variables i, j, and k in Equation (10) assumes an integer value, its domain can be extended to the set of rationals. For instance, an m-dimensional domain of processes is now embedded in an m-dimensional vector space over the rationals. From now on, we may refer to the m-tuple of values of recursion variables identified with a process as a vector. As suggested by [19], a data dependency vector plays an important role in mapping algorithms to parallel processors. Since the vector addition is defined and the difference of two vectors takes on an exact meaning, a data dependency vector can be formally defined:

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

The three data dependency vectors that appear in System (10) are

$$\mathbf{d}_1 \stackrel{\text{def}}{=} (1,0,0)^T,\ \mathbf{d}_2 \stackrel{\text{def}}{=} (0,1,0)^T,\ \mathbf{d}_3 \stackrel{\text{def}}{=} (0,0,1)^T.$$

6.3 Basis Communication Vectors

As described above, each program defines a set of data dependency vectors; on the other hand, each network topology defines a set of *basis communication vectors*. For instance, in an n-dimensional hypercube, a processor has n connections to its nearest neighboring processors. Each of the n communication vectors (one for each connection), has $n+1$ components: the first n ones indicating the movement in space, and the last in time, which is always positive (counting invocations). These n-communication vectors, together with the communication vector $[0,0,\ldots,0,1]$, representing the processors's communication of its current state to its next state, form the basis communication vectors. In an n dimensional network, there can be more than one set of basis communication vectors. Taking a two-dimensional hexagonal network as an example, a diagonal connection has a communication vector $[1,1,1]$. The set of vectors $\{[1,0,1],[0,1,1],[1,1,1]\}$ serves as bases as well as the set $\{[1,0,1],[0,1,1],[0,0,1]\}$. Any n-dimensional network which has nearest neighbor connections and is regularly connected and indefinitely extensible can be obtained by the subgroups of the symmetry groups Lin and Mead [16]. All such possible two dimensional networks are shown in Figure 3.

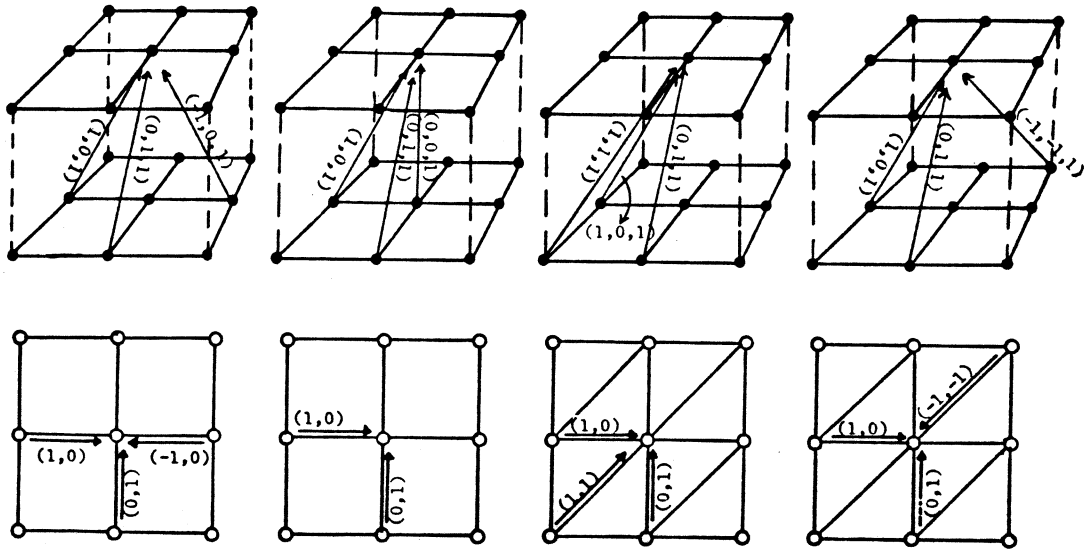

Figure 3: All the possible two dimensional regularly connected network (the bottom row), and the corresponding process structures in the 3-dimensional space-time (the top row).

6.4 Uniformity of a Parallel Algorithm

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

Definition 6.2 (Uniformity) *A uniform algorithm is one in which a single set of basis vectors* $\mathbf{b}_j$, $1 \leq j \leq m$ *can be chosen so as to satisfy the following mapping condition: every data dependency vector* $\mathbf{d}_i$ *appearing in the algorithm can be expressed as a linear combination of the chosen basis vectors with non-negative coordinates:*

$$\begin{aligned} &\textit{there exist } \mathbf{b}_j, 1 \leq j \leq m \textit{ such that} \\ &\mathbf{d}_i = \alpha_1 \mathbf{b}_1 + \alpha_2 \mathbf{b}_2 + \cdots + \alpha_m \mathbf{b}_m, \textit{ for } 1 \leq i \leq k, \textit{ where } \alpha_j \geq 0 \end{aligned} \tag{11}$$

6.5 Motivation for the Mapping Condition

The mapping condition is motivated by the possibility of using as the space-time mapping a linear mapping from the basis dependency vectors to the basis communication vectors. Each basis communication vector $\mathbf{e}_j$ corresponds to a nearest neighbor communication on a network of processors, and its time component is always 1. When the mapping between the two sets of basis vectors is determined, then the communication vector $\mathbf{c}_i$, to which a data dependency vector $\mathbf{d}_i$ corresponds is also determined, i.e., $\mathbf{c}_i = \sum_{j=1}^{m} \alpha_j \mathbf{e}_j$ if $\mathbf{d}_i = \sum_{j=1}^{m} \alpha_j \mathbf{b}_j$.

If a data dependency vector $\mathbf{d}_i$ has a term with a negative coordinate α_{j1} in its linear combination, then $\alpha_{j1}\mathbf{e}_j$ represents a communication that takes negative time steps, a situation that is not feasible in any physical implementation.

It can be shown that uniformity is the necessary and sufficient condition for the existence of a linear function as space-time mapping, and that a set of optimal basis dependency vectors $\{\mathbf{b}_j\}$ can be generated so that the maximum path length of the resulting communication vectors $\mathbf{c}_i$ for all i is the shortest over all possible choices of $\{\mathbf{b}_j\}$. The procedure of uniformity test and the generation of an optimal set of basis dependency vectors has a time complexity polynomial

difference vectors	CV1	CV2	CV3	CV4
$\mathbf{d}_1 \stackrel{\text{def}}{=} (1,0,0)$	$[1,0,1]^T$	$\mathbf{c}_1 \stackrel{\text{def}}{=} [1,0,1]^T$	$[1,0,1]^T$	$[1,0,1]$
$\mathbf{d}_2 \stackrel{\text{def}}{=} (0,1,0)$	$[0,1,1]^T$	$\mathbf{c}_2 \stackrel{\text{def}}{=} [0,1,1]^T$	$[0,1,1]^T$	$[0,1,1]$
$\mathbf{d}_1 \stackrel{\text{def}}{=} (0,0,1)$	$[0,0,1]^T$	$\mathbf{c}_3 \stackrel{\text{def}}{=} [1,1,1]^T$	$[-1,-1,1]^T$	$[-1,0,1]$
linear mapping*	T_1 $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	T_2 $\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	T_3 $\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$	T_4 $\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
MM	obvious	[23]	[22]	
LU	[9,8]	[7]	[22]	

Table 1: Data dependecy vectors, communication vectors, linear mappings, and systolic algorithms for matrix multiplication and LU decomposition.

of the dimensionality of the process structure, which is usuallly a small constant independent of the problem size. The results mensioned in this paragraph will be presented in a sequel.

In the case of a non-uniform program, more than one set of basis data dependency vectors must be chosen so as to satisfy the mapping condition, and the space-time mapping may become non-linear. In this case, the inductive mapping procedure becomes necessary.

Examples of using such a simple procedure to find linear mappings of processes to parallel architectures for LU decomposition, dynamic programming, and array multipliers can be found in [6,7,2,3]. Most of the systolic algorithms reported in the literature can be obtained this way, and new systolic algorithms are discovered due to the ability to generate systematically all possible sets of basis communication vectors.

6.6 Obtain Linear Mappings for Uniform Algorithms

For matrix multiplication, let the basis data dependency vectors be the three data dependency vectors themselves. Clearly, they satisfy the mapping condition and System (10) is a uniform algorithm.

For a uniform algorithm, each linear function that maps the basis dependency vectors to basis communication vectors determines a parallel implementation. Since there are numerous sets of basis communication vectors, many implementations using different network topology with different control and data flow may result.

Thus for each set of basis communication vectors obtained from the subgroup enumeration, we immediately obtain the linear mapping from the basis dependency vectors to them, and in turn the mapping T from processes to invocations of processors.

Listed in Table 1 are the basis communication vectors for matrix multiplication in System (10), four different sets of communication vectors (CV1, CV2, CV3, CV4) and the resulting systolic algorithms. Also listed are the systolic algorithms for LU decomposition derived using the same method.

*In the matrix notation where all dependency and communication vectors are written as column vectors.

7 Space-time Recursion Equations

Proposition 7.1 *The space-time mapping* $T_2 \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ *maps every process* (i, j, k) *(written as a column vector) of* P *to an invocation* $[x, y, t]$ *(written as a column vector), or in a functional notation:*

$$(x, y, t) = f(i, j, k) = (i + k, j + k, i + j + k) \text{ where } 1 \leq x \leq 2n, 1 \leq y \leq 2n, 2 \leq t \leq 3n.$$

The inverse mapping from the image of f *to the domain of processes, denoted by* f^{-1} *is*

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

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

Figure 4 shows the process structure for the matrix multiplication in the original coordinate system (from System (10)), and the new axis using mapping T_2. In the case where the communications to the outside world from a network of processors are limited only to those via its boundary processors, the inputs must reside outside the boundaries initially, and then be shifted into the network. For this example, we know where the inputs should appear in the process stucture from System (10), as shown in Figure 4. To figure out where they should be outside the network initially, the inputs $A(i, k)$, $B(k, j)$, and $C(i, j)$ are shifted out from inside the boundaries of the process structure in the directions opposite to $[0, 1, 1]$, $[1, 0, 1]$, and $[1, 1, 1]$ respectively, until all inputs are outside of the network. The two-dimensional systolic network obtained by mapping T_2 and the appropriate initialization of its input streams are depicted in Figure 5.

Formally, a system of *space-time recursion equations* can be obtained from System (10) by algebraic transformations using the space-time mapping f. The procedure for space-time mapping is summarized in Figure 6. Various other features for each algorithm, such as the number of processes and time steps needed, the latency, the utilization rate of each processor, etc., can also be obtained by using f and f^{-1}.

This new program can now be interfaced to silicon compilers to produce special-purpose VLSI designs, or else undergo another stage of transformations that map the algorithm using a virtual network (a network with ideal size and connectivity) to a machine with a given architecture and size. For instance, a **Crystal** program using a d-dimensional virtual network can be mapped to hypercube machines by employing the scheme of binary reflected Grey encoding. Some code optimization that is machine-characteristic dependent might be needed at this stage to balance the amount of computations and communications. The steps of transformation at this stage are shown in Figure 7.

8 Concluding Remarks

Crystal provides a simple, straightforward, but powerful way of defining computational problems. When interpreted by **Crystal**, the definition naturally discloses a parallel algorithm where the communications between processes and the computation within an individual process become clear. The **Crystal** programming environment takes such definitions, analyses and performs the necessary transformations to yield algorithms having bounded degrees of fan-in and fan-out and using only local communications. The resulting algorithms go through the process of space-time mapping to yield programs with pipelining automatically incorporated.

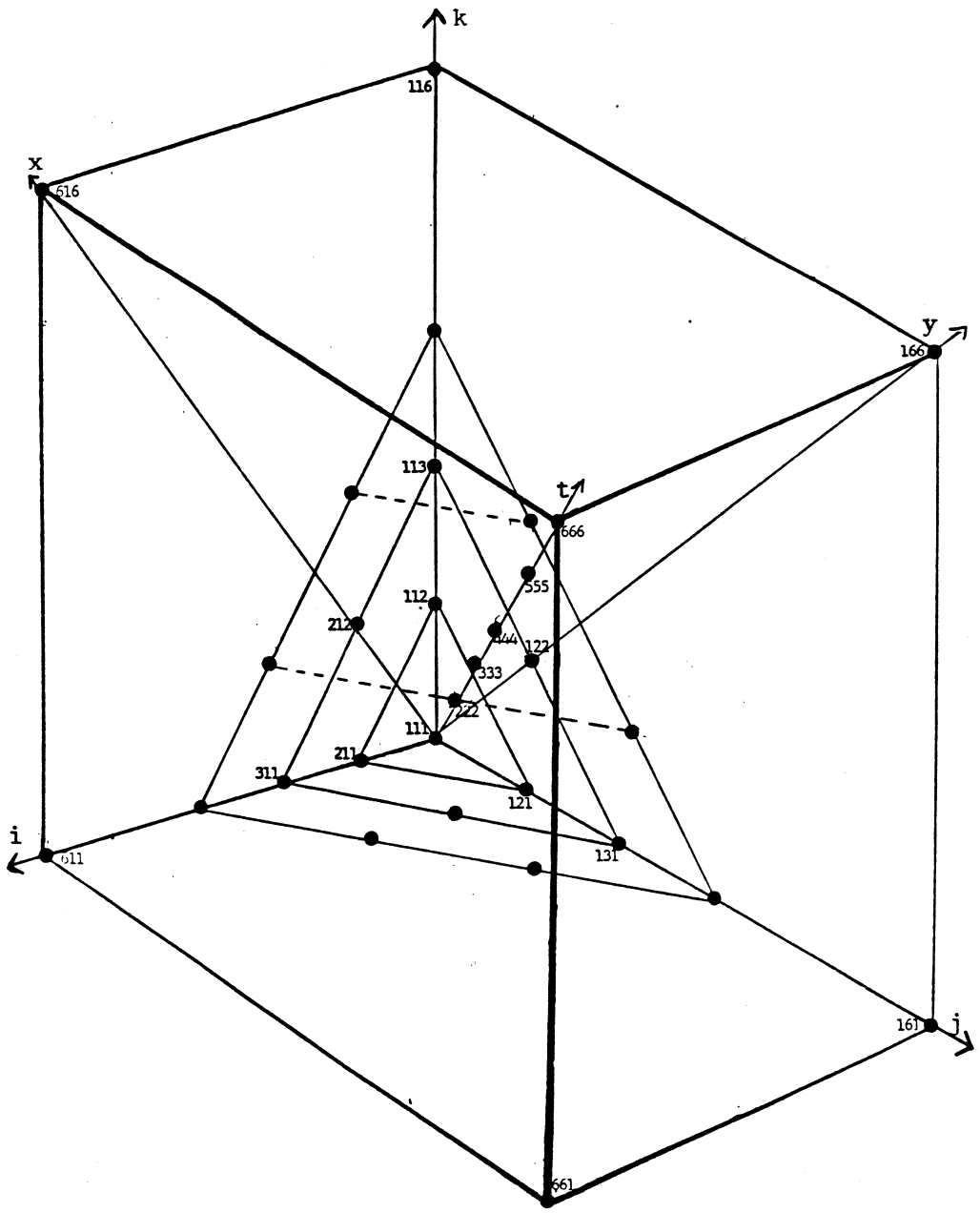

Figure 4: Process structure in the original coordinate system, and the axis of the new coordinate system.

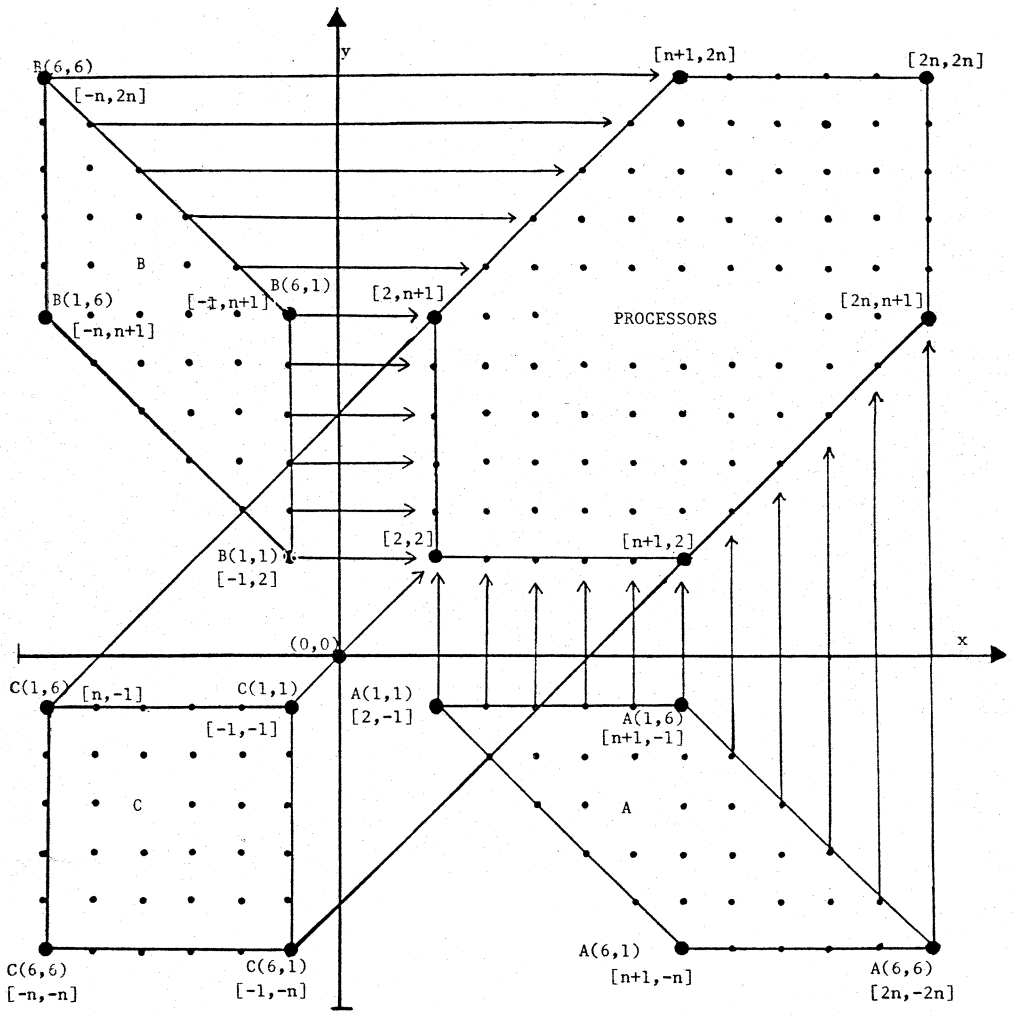

Figure 5: The systolic network using basis communication vectors CV2.

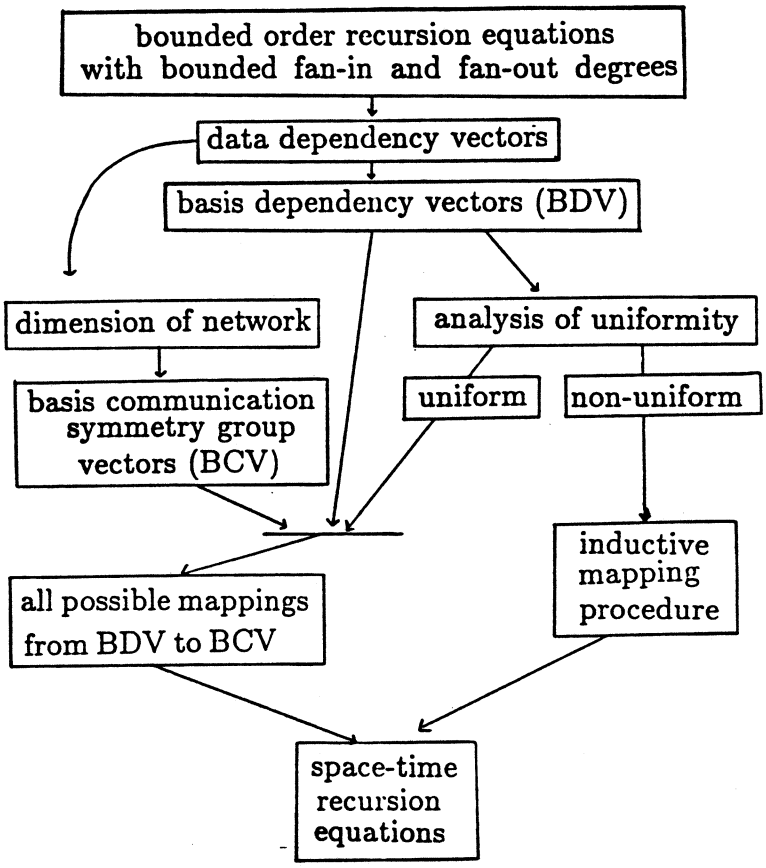

Figure 6: Incorporating pipelining by space-time mapping.

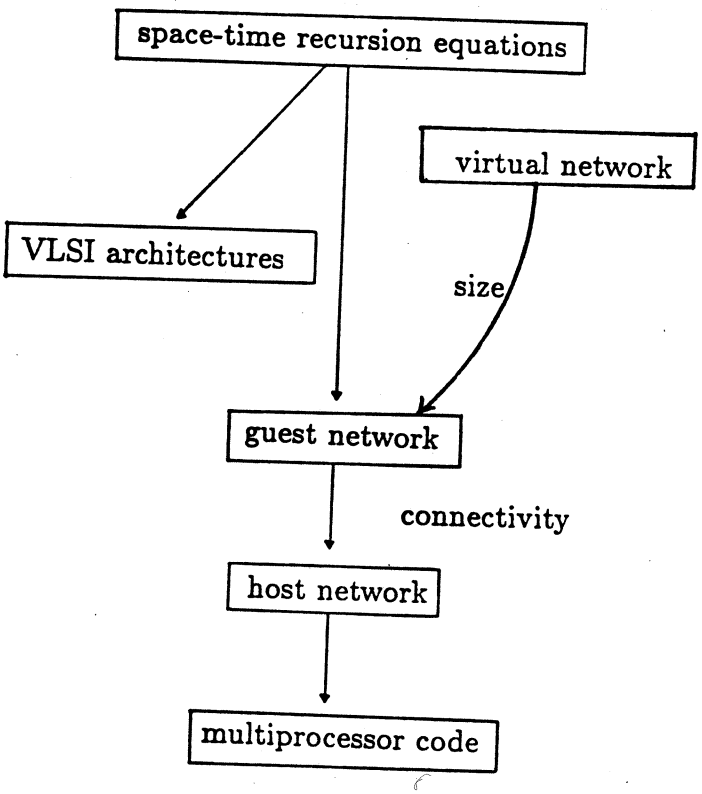

Figure 7: Mapping parallel algorithms to multiprocessors.

Thus, based on the language **Crystal**, a unified theory for understanding and generating systolic computation comes into existence.

From a programmer's point of view, **Crystal** is easy to use because the programmer does not need to specify explicit communications in a program; the specification is a functional definition that is natural to the problem rather than tailored to machine execution. The correctness of a **Crystal** program is easy to ensure. First of all, we start with problem definition whose correctness is self-evident and scientifically estabilished. Secondly, **Crystal** provides high-level functions that capture efficient computations and communications of many processors as a whole. The problem definition is then transformed by employing these high-level functions to produce programs with guranteed correctness as well as efficiency.

Crystal encourages large scale parallelism, but at the same time allows a complex system to be specified by a hierarchy of parallel sub-systems by composition and abstraction. The **Crystal** compiler generates all possible space-time implementations from a functional and deterministic definition by using a set of powerful synthesis methods, as opposed to allowing programmers to use non-deterministic programming constructs to specify possible alternative implementations which are often error-prone.

The high level notation and algebraic properties the language **Crystal** enjoys allow programmers to use **Crystal** not only as a means to interact with computer systems, but as a way of communciation among themselves. On the other hand, they make **Crystal** amenable to algebraic manipulation; thus all algorithm transformations described above can be carried out mechanically.

Last but not least is the portability issue. Notice that in this approach to parallel processing, successive transformations are performed on programs all written in the same language — **Crystal** — including the code optimization step which takes the machine characteristics as parameters. Only at the last code generation step do different machines require different code generators and routing schemes. Such a high level of portability is especially important in an age where larger, faster and ever more different machines are built.

References

[1] R. E. Bryant. *A Switch-Level Simulation Model for Integrated Logic Circuits.* PhD thesis, Massachusetts Institute of Technology, March 1981.

[2] M. C. Chen. *Automatic Generation of VLSI Architectures: Dynamic Programming Solver.* Technical Report 455, Yale University,, Febuary 1986.

[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*, page , October 1985.

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

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

[6] M. C. Chen. *A Synthesis Method for Systolic Designs.* Technical Report 334, Yale University, January 1985.

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

[8] J. Delosme and Morf M. Scattering arrays for matrix computations. In *Proceedings of SPIE, Real time signal processing IV*, pages 74–91, August 1981.

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

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

[11] Ming-Deh A. Huang. Solving some graph problems with optimal or near-optimal speedup on mesh-of-trees networks. In *Proceedings of FOCS*, pages 232–240, IEEE, October 1985.

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

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

[14] G.-J. Li and Wah B. W. The design of optimal systolic arrays. *IEEE Transactions on Computer*, C-34(1):66–77, January 1985.

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

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

[17] W. L. Miranker. Spacetime representations of computational structures. In *Computing*, pages 93–114, 1984.

[18] Dan. I. Moldovan. Advis: a software package for the design of systolic arrays. *Proceedings of ICCD*, 1984.

[19] Dan I. Moldovan. On the design of algorithms for vlsi systolic arrays. In *IEEE Transaction on Computer*, 1983.

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

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

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

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