A PARALLEL LANGUAGE AND ITS COMPILATION TO MULTIPROCESSOR MACHINES OR VLSI

Marina C. Chen

Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

A language **Crystal** and its compiler for parallel programming is presented. The goal of **Crystal** is to help programmers in seeking efficient parallel implementations of an algorithm, and managing the complexity that might arise in dealing with hundreds of thousands of autonomous parallel processes. In **Crystal**, a program consists of a system of recursion equations and is interpreted as a parallel system. **Crystal** views a large complex system as consisting of a hierarchy of parallel sub-systems, built upon a set of **Crystal** programs by composition and abstraction. There is no mention of explicit communications in a **Crystal** program. The **Crystal** compiler automatically incorporates pipelining into programs, and generates a parallel program that is optimal with respect to an algorithm. Each optimizing compiler, targeted for a particular machine, determines the appropriate granular size of parallelism and attains a balance between computations and communications. Based on the language, a unified theory for understanding and generating any systolic design has been devised and it constitues a part of the compiler.

1. Introduction

The effective exploitation of the parallelism provided by multiprocessor machines or special purpose VLSI designs clearly relies on a suitable parallel programming language and a powerful compiler to help in (1) seeking efficient parallel implementations of an algorithm, and (2) managing the complexity that might arise in dealing with hundreds of thousands of autonomous parallel processes. Toward these two goals, several issues concerning language arise:

1. What kind of languages support expression of concurrency implicitly and free programmers from specifying explicit communications?
2. Balanced communications and computations are central to an efficient parallel implementation of a program. Does a language allow the relationship between the two to be clearly reflected in a program so that an optimizing compiler can perform trade-offs and come up with an efficient implementation?
3. Is the language general purpose? Does it allow a network topology to be easily expressed?
4. Does a language provide constructs that encourage the exploitation of large-scale parallelism, and simultaneously, promote the clarity of programs and ease of ensuring correctness?

Crystal*, along with its compiler, is an attempt to provide answers to these questions. In **Crystal**, a program consists of a system of recursion equations (e.g. in Manna[15]) similar to those appearing in applicative languages, but it is interpreted completely differently from the applicative interpretation (or its parallel version). **Crystal**'s parallelism is shown to be more efficient than the applicative parallelism. There is no mention of explicit communications in a **Crystal** program. The **Crystal** compiler automatically incorporates pipelining into programs, and generates a parallel program that is optimal with respect to an algorithm. Each optimizing compiler, targeted for a particular machine, determines the appropriate granular size of parallelism and attains a balance between computations and communications.

Compared with other high-level programming environments that allow parallel implementations such as Shpiro's systolic programming in Concurrent Prolog [20] and Snyder's Poker programming environment, **Crystal** goes one step further by automating the process of decomposing computations to parallel processes. Efficient parallel programs are generated from algorithms by taking into account the dominant cost in parallel processing: the cost of communications. Moreover, a generated parallel program is expressed in terms also of recursion equations, the same language as that for an algorithm. Hence in **Crystal**, the equivalence of an algorithm and its parallel implementation can be shown formally and guaranteed mechanically.

In Section 2, the language **Crystal** and its interpretation are introduced. The second stage of **Crystal**'s compilation process, space-time mapping, is presented in Section 3. Space-time mapping generates programs that are systolic, in other words, it syntheizes systolic designs from an algorithm specified in a high level language. The third stage of compilation concerns with finite sizes of machines. Techniques for an optimizing compiler and the first stages of **Crystal**'s compilation process are discussed in Section 4. In the last section, a few efficient parallel implementations of **Crystal** programs are mentioned. Finally, various language issues are revisited to review the capablility of **Crystal**.

*It stands for Concurrent Representation of Your Space-Time Algorithms [2], a language for parallel computing. It bears no relationship with the distributed operating system Crystal Nugget [9], nor with the VLSI timing verifier Crystal [17], nor the Crystalline software on the Cosmic Cube [19], nor with the character portrayed by Linda Evans on the popular television soap opera *Dynasty*.

2. The Parallel Language Crystal

2.1. A high level parallel program

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):

$$C(i,j) = \begin{cases} i = 1 \rightarrow 1 \\ i > 1 \rightarrow \begin{cases} i > j \rightarrow C(i-1,j) \\ i = j \rightarrow C(i-1,j) + 1 \\ i < j \rightarrow C(i-1,j) + C(i,j-i). \end{cases} \end{cases} \tag{2.1}$$

2.2. Applicative parallelism

Equation (2.1) is a recursive definition that is common in any applicative programming language. Its applicative interpretation can be illustrated by a tree as shown in Figure 1. Certainly, in such an applicative view, the calls, or invocations of the recursive definition $C(i,j)$ need not be made sequentially; parallelism can be extracted from, for instance, the parallel evaluation of the left branch $C(i-1,j)$ and the right branch $C(i,j-i)$ of the last expression in the equation.

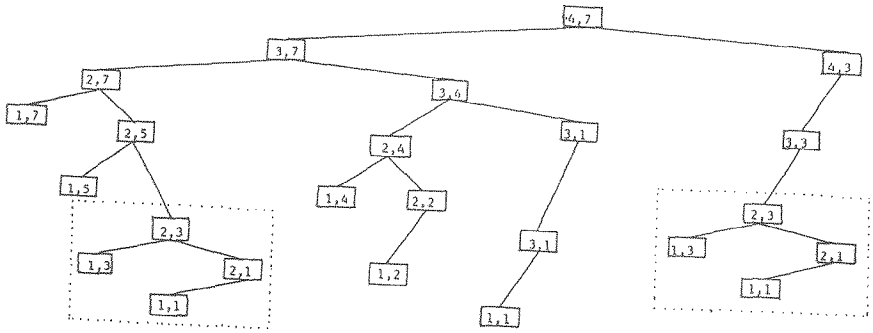

Figure 1: The applicative interpretation of Equation (2.1) for $C(4,7)$.

2.3. Crystal's parallelism

In **Crystal**, however, a completely different interpretation is given to the equation, and parallism of a different kind is introduced. Instead of being considered as actual parameters at a call, the pair (i,j) is considered as an index pair for a process in an ensemble of parallel processes. For each pair (i,j) in the set $D \stackrel{\text{def}}{=} \{(i,j) : 0 < i \leq m, 0 < j \leq k\}$, there corresponds a process. The process structure of this program can be seen as a DAG (Directed Acyclic Graph), which is shown in Figure 2. The DAG depicts the data dependency of each $C(i,j)$ for all $(i,j) \in D$. It consists of nodes, where each node corresponds to an index pair (i,j) in the set D, and directed edges, each of which comes out of a node whose index pair appears on the right hand side of Equation (2.1), and goes into a node whose index pair appears on the left hand side of the equation. The directed edges of the DAG define the data dependency relation of the algorithm. We say that a node u precedes v ($u \prec v$), or v depends on u ($v \succ u$), if there is a directed edge from u to v. The transitive closure of this relation on all such pairs is a partial order, and there is no infinite decreasing chain from any pair (i,j). Those nodes that have no incoming edges are called *sources*.

In **Crystal**, processes are parallel in nature. A computation starts at the sources which are properly initialized, and is followed by other processes each of which starts execution when all of its required inputs, or dependent data, become available. If the applicative interpretation is demand-driven, where a call $C(i,j)$ is invoked only when it is needed by its parent node in the tree, then **Crystal**'s view is data-driven, where processes at the sources of the DAG start autonomously after initialization, and the results of their computation in turn cause other processes to start.

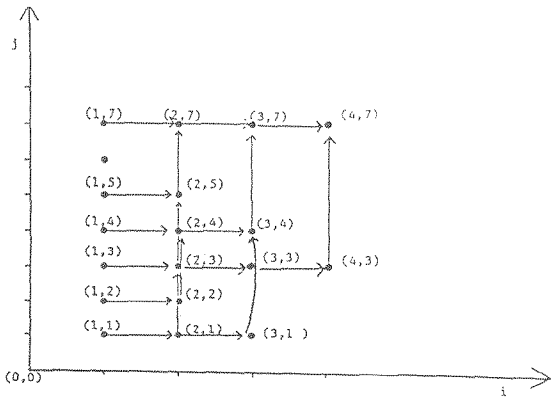

Figure 2: The DAG for the problem of partitions of an integer

2.4. Advantage of Crystal's parallelism

Is there any advantage to **Crystal**'s interpretation of recursion equations? The asymptotic complexity for computing $C(m,k)$ by applicative parallel evaluation amounts to $O(m+k)$ for the time requirement and $O((\frac{k}{\min(k-1,m)})^m)$ for the space requirement. The time complexity is obtained from counting the longest path from the root of the tree $C(m,k)$ to a leaf node, i.e., $C(1,*)$, and the space complexity is obtained by noticing that if $k < m$ then $C(m,k) = C(k-1,k) + 1$ and then from the sum $1 + \frac{k}{2} + (\frac{k}{3})^2 + \cdots + (\frac{k}{m})^{m-1}$ for $k > m$, where each term $(\frac{k}{i})^{i-1}$ is the asymptotic space complexity for computing the right branch of $C(i,k)$. In contrast, a naive implementation of **Crystal**'s parallelism requires only $k \times m$ number of processors (the number of nodes in the DAG) and $k+m$ time steps (the longest path from a source to node (m,k)). This implementation is to use one processor for each process in the DAG. Based on its data-driven view of parallelism, **Crystal**'s compiler further increases the resouces utilization by reducing the number of processors needed from, say $O(n^k)$ to $O(n^{k-1})$ without increasing the time complexity, where n is the problem size and k some constant.

When a computing system becomes space limited due to problem sizes, the dramatic savings in the space requirements of **Crystal**'s parallelism over that of the applicative parallelism translates to savings in time, because invocations must be sequentialized under such limited circumstances. The exponential space requirement in the applicative parallel evaluation stems

from the evaluations of the same sub-tree by multiple branches, independent of one another. This wasteful situation may be avoided by checking for duplicated evaluations at run time, but not without a significant cost. It might also be avoided if a programmer were to rewrite the recursive program using iterative constructs instead, but at the expense of the conceptual elegance provided by recursion equations.

The advantage of **Crystal**'s parallelism goes beyond a particular example. As stated in Section 2, **Crystal** uses a style of recursion which is course-of-values recursion with unbounded minimalization, hence any program can be written in the above style. The more elements in the process structure, the more the parallelism. In contrast, the only parallelism that can be gained from the applicative scheme is by parallel evaluations of two or more independent branches due to the command-driven interpretation, which always requires an exponential number of processes with respect to the problem size. The growth is limited only in the case of a single branch, but then there is no parallelism that can be extracted. To utilize applicative parallelism, one would like to encourage more branches in a program to allow more parallel evaluations, yet this very act causes quicker exponential growth in the number of parallel processes, which in turn forces them to be sequentialized due to the inevitable space limitation in physical realilty. Such dilemma is a fundamental flaw in the applicative parallel interpretation scheme. In contrast, **Crystal**'s interpretation scheme results in efficent parallelism as well as retains the elegance of functional programming.

2.5. Comparison with CSP-like Languages

Many languages proposed for parallel programming, such as Communication Sequential Processes (CSP) [11] or Occam [12], are based on imperative seqential programming languages augmented with communication commands for expressing concurrency explicitly. There is no doubt that such a language is general purpose and flexible enough for programmers to specify any desirable parallelism; the question is whether it supports parallel programming at a suitable level. Taking any program written in CSP, for instance, one can see very clearly the computation of each individual process and, locally, how it explicit communicates with others. However, such a program presents no glimpse of what the global structure of parallelism might be.

In fact, due to the inherent asynchronous style in which communication commands are used, it is often necessary to simulate the program to see what a process structure is. The lack of global structure makes programs written in such language error prone and difficult to debug, a situation analogous to assembly language programs in sequential programming, except that the issue of correctness becomes a even more severe problem due to the complexity of large scale parallelism. Any sequential language augmented with a set of communication commands serves well as a parallel language, but only at the object language level for a compiler to generate code in, rather than be used by programmers directly. For instance, a **Crystal** compiler targeted for Intel's iPSC hypercube multiprocessor uses the programming language C augmented with iPSC's communication commands as an object language.

2.6. Comparison with data flow languages

The computation model of **Crystal** differs from those of the data flow languages [8] in that the flow graph (the process structure) is a directed acyclic graph (DAG), where each node in a DAG represents an abstract process (invocation of a function). In contrast, the flow graph in data flow languages is often cyclic where iterative loops are represented by cycles, and each node in a flow graph represents some operation or functional unit (a processor) rather than an invocation. Tokens that flow through the graph are then necessary to model the invocations. By comparison, **Crystal**'s model of computation is at an abstract level rather than being bound to some implementation as in the model of data flow languages. Furthermore, **Crystal**'s synthesis method finds one or more mappings of processes (nodes in the DAG) to processors (organized in some network topology) to yield efficient parallel programs of an algorithm.

Besides of its independence from implementations, **Crystal**'s style of programming with data streams defined over a process structure encourages a much larger scale of parallelism than that occurs in data flow systems [10].

2.7. Recursion equations

In Equation (2.1), we call i and j recursion variables and C a functional variable of the equation. In general, a parallel program is defined by a system of recursion equations

$$\begin{aligned} F_1(\mathbf{v}) &= \phi_1(F_1(\tau_{11}(\mathbf{v})), F_2(\tau_{12}(\mathbf{v})), \ldots, F_n(\tau_{1n}(\mathbf{v}))) \\ F_2(\mathbf{v}) &= \phi_2(F_1(\tau_{21}(\mathbf{v})), F_2(\tau_{22}(\mathbf{v})), \ldots, F_n(\tau_{2n}(\mathbf{v}))) \\ &\ldots \\ F_n(\mathbf{v}) &= \phi_n(F_1(\tau_{n1}(\mathbf{v})), F_2(\tau_{n2}(\mathbf{v})), \ldots, F_n(\tau_{nn}(\mathbf{v}))), \end{aligned} \tag{2.2}$$

where

- $\mathbf{v} \stackrel{\text{def}}{=} [v_1, v_2, \ldots, v_n] \in A$, A is a discrete structure, which is a Cartesian product $A_1 \times A_2 \times \cdots \times A_n$ where each A_i, say, is a subset of the set of integers, or the set of rationals, and $(A_i, \sqsubseteq)$ is a flat lattice, where "$\sqsubseteq$" is the approximation order [18].
- function τ_{ij} maps from A to A.
- functional variables $F_1, F_2, \ldots, F_n$, range over continuous functions mapping from $(A, \sqsubseteq)$ to $(D, \sqsubseteq)$, where D is some value domain such as the set of integers, reals, etc., and $(D, \sqsubseteq)$ is a flat lattice.
- functions $\phi_1, \phi_2, \ldots, \phi_n$, are continuous functions over the value domains $(D, \sqsubseteq)$. Thus (2.2) is a system of fixed-point equations, and on its right hand side,

  $$\lambda(F_1, F_2, \ldots, F_n).\lambda\mathbf{v}.[\phi_1(\cdots), \phi_2(\cdots), \ldots, \phi_n(\cdots)]$$

  is a continuous function mapping from E^n to E^n, where $E \stackrel{\text{def}}{=} [(A, \sqsubseteq) \to (D, \sqsubseteq)]$, the domain of continuous functions from $(A, \sqsubseteq)$ to $(D, \sqsubseteq)$.

Such a system of equations embodies a parallel computation on a discrete structure A with multiple data streams F_i, processing functions ϕ_i, and communication functions τ_{ij}, where $1 \le i, j \le n$. The discrete structure A is in general some n-dimensional space. It ranges from the degenerate case (0 dimensional), where a piece of straight code (without any iteration or recursion construct) is executed on a uniprocessor, to a two-dimensional case with a systolic program running on a linear array, or to a multi-dimensional case with programs running on butterfly, tree, or hypercube machines. A set of communication functions is often associated with each particular process structure A. Each processing function ϕ_i is a composition of base functions (e.g. "+" in (2.1)) from some value domain $(D_1, \sqsubseteq)$ to another value domain $(D_2, \sqsubseteq)$.

The outermost base function in the composition is often a "case" function, as is the one with four cases (flattened from the nested conditionals) in Equation (2.1). Each equation in (2.2) can be written structurally in more detail as k cases:

$$F_i(\mathbf{v}) = \begin{cases} p_{i1}(F_1(\tau_{i11}(\mathbf{v})), F_2(\tau_{i12}(\mathbf{v})), \ldots, F_n(\tau_{i1n}(\mathbf{v}))) \rightarrow \\ \quad \phi_{i1}(F_1(\tau_{i11}(\mathbf{v})), F_2(\tau_{i12}(\mathbf{v})), \ldots, F_n(\tau_{i1n}(\mathbf{v}))) \\ p_{i2}(F_1(\tau_{i21}(\mathbf{v})), F_2(\tau_{i22}(\mathbf{v})), \ldots, F_n(\tau_{i2n}(\mathbf{v}))) \rightarrow \\ \quad \phi_{i2}(F_1(\tau_{i21}(\mathbf{v})), F_2(\tau_{i22}(\mathbf{v})), \ldots, F_n(\tau_{i2n}(\mathbf{v}))) \\ \ldots \\ p_{ik}(F_1(\tau_{ik1}(\mathbf{v})), F_2(\tau_{ik2}(\mathbf{v})), \ldots, F_n(\tau_{ikn}(\mathbf{v}))) \rightarrow \\ \quad \phi_{ik}(F_1(\tau_{ik1}(\mathbf{v})), F_2(\tau_{ik2}(\mathbf{v})), \ldots, F_n(\tau_{ikn}(\mathbf{v}))) \end{cases}$$

where $p_{i1}, \ldots, p_{ik}$ are boolean predicates.

The number of cases k depends on the homogeneity of the processing functions and the uniformity of the communication functions over structure A. Since the predicates are functions of F_i's, both the processing and communication (or data flow) of each element in A might be data dependent and change dynamically during a computation. The classification of parallel machines can be borrowed here: For the single instruction multiple data (SIMD) type of machine architectures, the processing and communication functions of all elements of the process structure are the same, hence the recursion equations are of a simple form. For the MIMD type of architectures, the recursion equations are usually composed of many cases.

Note that the system (2.2) is in the form of simultaneous, course-of-values recursion, which is primitive recursive. Thus unbounded minimalization is a construct for specifying the computation of general recursive functions. This construct is particularly useful in describing the process of relaxation in various simulation tasks.

2.8. Correctness of a Crystal program

In Crystal, correctness is easy to ensure, as the behavior of a program is the least fixed point of the system of recursion equations which can be verified by structural induction [1] on the DAG.

2.9. Structured programming in Crystal

Similarly to any sequential programming language, Crystal provides modular constructs for structured programming. Processing functions and communication functions can be defined separately, also as recursion equations, and then used in a program. In Crystal's view, a large complex system consists of a hierarchy of parallel sub-systems. At a given level, each sub-system is modularized depending on what is natural to a particular problem. The system of recursion equations defined for a sub-system at one level is used as a function (its least fixed-point) at the level above it. In such a philosophy of computing, computations are defined independently of how they are going to be implemented — sequentially or in parallel. A Crystal compiler, specialized for each target machine, is responsible for determining the level beyond which parallel implementation is used and the level below which a sequential implementation of Crystal's interpretation of recursion equations is used. Such closure in composition of Crystal programs and capability in alleviating programmers from implementation details are essential to complex systems with massive parallelism.

2.10. Space-time recursion equations

From an implementation point of view, a process $\mathbf{v}$ in a recursion equation (2.2) will be mapped to some physical processor s during execution, and once it is terminated, another process can be mapped to the same processor. 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 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 $[s, t]$, telling when (which invocation) and where (in which processor) it is executed.

How shall the name of a process be related to its alias? The key to an efficient implementation of a parallel program is to find a suitable one-to-one function that maps a name to its alias. Once such a function is obtained, it can be substituted into the program to yield another system of recursion equations which has the property that (1) the time component s must always be non-negative, and (2) the time component of any invocation on the right hand side of an equation must be strictly less than the time component of the invocation on the left hand side of the equation. Such equations are called *space-time* recursion equations (STREQ) [3] and the function a *space-time mapping*. A system of space-time recursion equations specifies completely what code each processor executes at each invocation (its processing function), how the processor communicates with other processors (its communication functions), how communications and computations are controlled locally at each processor (its predicates), how each processor should be initialized (its specification at time zero), and how input streams should be supplied (its specification at the boundary processors of a multiprocessor machine).

3. Compilation Method

The goal of the Crystal compiler is to generate, from a given algorithm or definition specified by a system of recursion equations, a system of space-time recursion equations that is optimal for executing on the target machine. In what follows, we call such a parallel implementation described by a system of space-time recursion equations a *design*. The procedure of Crystal's compilation method is shown in Figure 6. In the following, the second stage of the compilation method, a systhesis method which automatically incorporates pipelining, is illustrated by deriving a design to solve the integer partition problem. An optimal design which uses an array of m processors and computes $C(m, k)$ in total $k + m$ time steps with latency m is generated.

3.1. Process structure and communication functions

As mentioned earlier, the process structure D is of two dimensions. The communication functions consist of $\tau_1(i, j) = (i - 1, j)$ and $\tau_{2i}(i, j) = (i, j - i)$. Note that for each different i, there is associated a different communication function τ_{2i}. This variation suggests that the partition problem may need to be solved by a design that is not quite regular.

3.2. Difference vectors

The discrete process structure can be embedded in a vector space over the rationals; thus each "process id" can be treated as a vector. A difference vector is defined to be the difference of (i, j), the left hand side index pair of Equation (2.1) and $\tau_1(i, j)$ or $\tau_{2i}(i, j)$, which appear on the right hand side of the equation. The difference vectors obtained from this equation are $(1, 0)$, and $(0, i)$ for $0 < i \leq m$.

3.3. Uniformity of a parallel program

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. A uniform program is defined to be:

Definition 3.1. A uniform program is one in which a single set of basis vectors can be chosen so as to satisfy the following mapping condition: each difference vector appearing in the program can be expressed as a linear combination of the chosen basis vectors with non-negative coordinates.

The mapping condition is motivated by the possibility of using as the space-time mapping a linear transform from the basis difference vectors to the basis *communication vectors*. A communication vector $[\Delta s, \Delta t]$ is just a difference vector of a system of STREQ, which always has a positive time component

Δt, and which represents a communication from a processor s to another processor $s+\Delta s$ in Δt time steps. If the mapping between the two sets of basis vectors is determined, then the mapping of each difference vector is also determined: its mapping is the same linear combination as itself, except in terms of the corresponding basis communication vectors. If a difference vector has a term with a negative coordinates in its linear combination, then the mapping of this term to space-time represents a communication that takes negative time steps, a situation that is not feasible in any physical implementation. In the case of a non-uniform program, more than one set of basis difference vectors must be chosen so as to satisfy the mapping condition, and the space-time mapping may turn out to be non-linear. An inductive mapping procedure described must be used to find the space-time mapping for non-uniform algorithms.

The program (Equation 2.1) for the partition problem is a uniform one, as all difference vectors appearing in Equation (2.1) can be expressed as linear combinations of basis vectors $(1,0)$ and $(0,1)$ with non-negative coordinates.

3.4. Basis communication vectors

Each network often comes with 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, 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 network as an example, connections other than the nearest neighbor connection are used, as in a hexagonal systolic array, where a diagonal connection has a communication vector $[1,1,1]$. Since in general there can be more than two processors along each dimension of a network, the communication vector $[1,1,1]$ is different from but as valid a basis communication vector as $[-1,-1,1]$. In fact, different sets of basis communication vectors result in different designs. The **Crystal** compiler finds all of these designs by enumerating all possible sets of basis communication vectors, so that the most suitable one for the target algorithm can be chosen.

For the partition problem, its process structure is two-dimensional and its design can be realized in a one-dimensional network of processors. The symmetry of a one-dimensional network can be described by the symmetry group

$$G \stackrel{\text{def}}{=} \{E(\text{identity}), R(180 \text{ degree rotation})\}.$$

Since the different directions of data flow at each processor have different symmetry properties, all possible directions of data flow at each processor can be obtained by enumerating the subgroups of G. This enumeration of possible directions of data flow at each processor by symmetry group [14] can be generalized to an n dimensional network. As shown in Figure 3, each subgroup is identified with a set of basis communication vectors that describes the data flow at each processor.

The sets of communication vectors, each of which corresponds to a subgroup of G, consist of $[1,1],[-1,1]\}$ (G itself as the subgroup), and $\{[0,1],[1,1]\}$ (subgroup $\{E\}$). All possible mappings from a set of basis difference vectors to a set of basis communication vectors are as follows, where the transpose of each of the basis vectors is defined as $\hat{i} \stackrel{\text{def}}{=} \begin{pmatrix}1\\0\end{pmatrix}$, $\hat{j} \stackrel{\text{def}}{=} \begin{pmatrix}0\\1\end{pmatrix}$, $\hat{r} \stackrel{\text{def}}{=} \begin{bmatrix}1\\1\end{bmatrix}$, $\hat{l} \stackrel{\text{def}}{=} \begin{bmatrix}-1\\1\end{bmatrix}$, and $\hat{t} \stackrel{\text{def}}{=} \begin{bmatrix}0\\1\end{bmatrix}$:

$T_1 : (\hat{i}\ \ \hat{j}) \mapsto (\hat{r}\ \ \hat{t})$, $T_2 : (\hat{i}\ \ \hat{j}) \mapsto (\hat{t}\ \ \hat{r})$, $T_3 : (\hat{i}\ \ \hat{j}) \mapsto (\hat{r}\ \ \hat{l})$

i.e. $T_1 = \begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}$, $T_2 = \begin{pmatrix}0 & 1\\ 1 & 1\end{pmatrix}$, $T_3 = \begin{pmatrix}1 & -1\\ 1 & 1.\end{pmatrix}$

Since the program is uniform, the mapping between the two basis sets gives the linear mapping from (i,j) to a space-time index pair $[s,t]$ immediately. For instance, linear transform T_1 results in the following mapping:

Proposition 3.1. *Linear transform T_1 results in a function f, $f(i,j) \stackrel{\text{def}}{=} [i, i+j]$ that maps process (i,j) to invocations of processors $[s,t]$, for $0 < i \le m$ and $0 < j \le k$.*

Since f is a 1 to 1 function, its inverse f^{-1} maps each invocation $[s,t]$ to a process (i,j). Let $Q[s,t] \stackrel{\text{def}}{=} C(i,j)$, then $Q = Cf^{-1}$. Algebraically, the following STREQ for the partition problem is obtained by substitution of the recursion variables i and j for s and t, difference vectors $(1,0)$ and $(0,i)$ for communication vectors $[1,1]$ and $[0,s]$, and renaming $C(i,j)$ as $Q[s,t]$ in Equation 2.1.

Proposition 3.2. *The space-time recursion equation for the partition problem is*

$$Q[s,t] = \begin{cases} s = 1 \rightarrow 1 \\ s > 1 \rightarrow \begin{cases} t < 2s \rightarrow Q[s-1, t-1] \\ t = 2s \rightarrow Q[s-1, t-1] + 1 \\ t > 2s \rightarrow Q[s-1, t-1] + Q[s, t-s] \end{cases} \end{cases}$$

where $0 < s \le m$ and $s < t \le k+m$ (3.1)

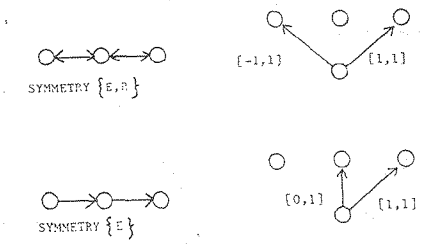

Figure 3: Symmetry groups and the sets of communication vectors that describe the data flow

3.5. Compiling STREQ to machine code or VLSI

The resulting design described by the STREQ is ready to be compiled locally to each processor.

3.5.1. Processor and time requirements

The design consists of a one-dimensional array of m processors which complete the computation in $k+m-1$ time steps, as indicated by the range of s and t. The network of processors and a few execution traces are shown in Figure 4.

3.5.2. Input ports and storage requirements

In the STREQ, if a communication vector has a non-zero space component, then the corresponding datum is an input such as $Q[s-1,t-1]$; otherwise it is a stored value such as $Q[s,t-s]$. Processor s $(0 < s \le m)$ must be incorporated with s registers because it must hold $Q[s,t]$ for s time steps. Thus a total of $O(m^2)$ distributed memory is used by this design. The "non-linearity" of this design shows up as the different numbers of registers needed by the processors. At each invocation, one

of the s registers is accessed and updated, and all of them are accessed by successive invocations in round-robin fashion.

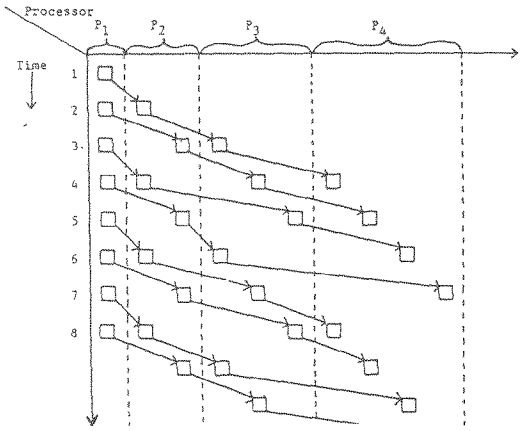

Figure 4: A systolic implementation for the partition problem

3.5.3. Control requirements

The guards of conditionals in the STREQ are implemented differently, depending on different target implementation media. In a multiprocessor machine, the "processor id" s is stored and an "*invocation* counter" keeps track of t until the condition $t > 2s$ is satisfied. In a VLSI implementation, however, it is too costly to store these two integers of length $\log(k+m)$ and perform arithematic operations on them. The following optimization of STREQ is needed.

3.5.4. Optimization performed on STREQ

For any guard that is not time dependent, such as $s = 1$, can be "hard-wired" in a design, however, a time dependent guard such as $t = 2s$ envolves keeping track of t. Any movement of a data stream takes time, hence it can be used to compute a time-dependent guard. We call such an added data stream a *control stream*. Equation 3.2 is a control stream which computes $t = 2s$. A better design can thus be obtained by replacing the expensive computations of the guards by transferring a one-bit control signal, and using a 2-bit control state to record which function should be performed when a guard is true. The optimized system of STREQ derived from Equation (3.1) is the following, where $R[s,t]$ is a data stream that carries the control signal, and $P[s,t]$ is the stored control state:

$$Q[s,t] = \begin{cases} s = 1 \rightarrow 1 \\ s > 1 \rightarrow \begin{cases} P[s,t] = 1 \rightarrow Q[s-1,t-1] \\ P[s,t] = 2 \rightarrow Q[s-1,t-1] + 1 \\ P[s,t] = 3 \rightarrow Q[s-1,t-1] + Q[s,t-s] \end{cases} \end{cases}$$

$$R[s,t] = \begin{cases} s = 1 \rightarrow \begin{cases} t = 2 \rightarrow 0 \\ t > 2 \rightarrow 1 \end{cases} \\ s > 1 \rightarrow R[s-1,t-2] \end{cases} \quad (3.2)$$

$$P[s,t] = \begin{cases} t = 0 \rightarrow 1 \text{ (initialization)} \\ R[s,t] = 0 \rightarrow 2 \\ R[s,t] = 1 \rightarrow 3 \\ \text{else} \rightarrow P[s,t-1] \end{cases} \quad (3.3)$$

Note that both control streams themselves do not use any time dependent guard, except for those that compare t with some constant. Central to an optimizing compiler for parallel programs is making trade-offs between communications and computations, as illustrated here and in Section 4 below.

3.6. Other designs for solving the partition problem

The design using mappings T_2 can be obtained similarly as above, and the space-time mapping function f is also a linear transform defined as $[s,t] = f(i,j) = [j, i+j]$ for $0 < i \leq m$ and $0 < j \leq k$. Such a design has k processors. For different i, the communication path at each processor s changes because the difference vector $(0,i)$ is mapped to a communication vector $[i,i]$ whose first component is non-zero. Figure 5(a) shows the execution traces for such a program. This systolic program can only be efficiently implemented by using switch networks due to the requirement for reconfigurable connections. Its direct implementation in VLSI is too costly in area because the connections provided for potential communications in each processor are of $O(m)$. One way to cut down on the number of connections is to decompose $[i,i]$ into compositions of $\hat{\tau} = [1,1]$ so that $C(i,j-i)$ is sent to the processor where $C(i,j)$ is computed via other processors. This systolic program, however, requires that the bandwith of each channel for communication be proportional to m, still not a good solution in terms of VLSI implementation. Figure 5(a) shows the execution traces of the modified design. The same implementation issues arise if it is to be implemented on a multiprocessor machine.

In summary, the time complexity of this systolic program is $k+m$, the same as derived from T_1, but the latency is k. The total number of processors required is k, and the total

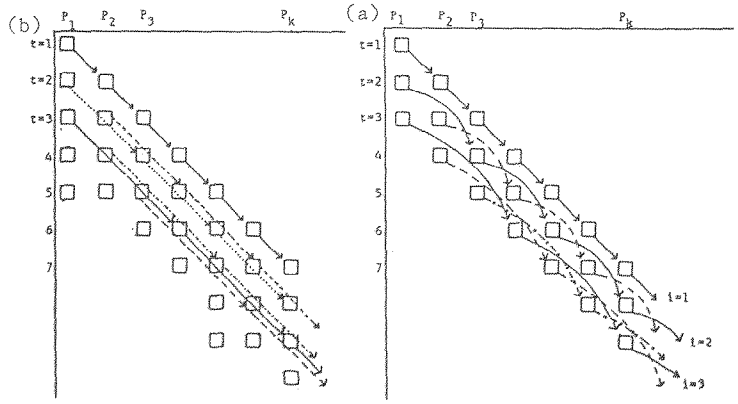

Figure 5: (a) The systolic program using linear transform T_2, which needs re-configurable connections. (b) A modified systolic design which needs channels of bandwidth $O(m)$

channel length (or channel bandwidth in the modified version) is of $O(m^2)$.

The program derived from mappings T_3 has problems similar to the one for T_2 since the difference vector $(0,i)$ is also mapped to a communication vector that is not stationary in space. The design is even less efficient then that obtained from T_2 because some of the invocations do not have processes mapping to them, hence they perform "no-op" instead of useful computations.

3.7. Limited number of processors and network dimensions

The space-time mapping above does not address the problems of a limited number of processors and a limited degree of connectivity at each processor. The first problem can be solved by modifying the space-time mapping to take the limitation into account. For example, for the partition problem, if the number of processor in the network is l, and $l < m$, the following space-time mapping f^* can be used:

$$[s,t] = f^*(i,j) \stackrel{\text{def}}{=} [i(\text{mod } l), i+j+(i/l)\times m],$$

where "/" is the integer division operation. This mapping based on f is obtained by (1) first "folding around" the network of size l to obtain a function $[s,t] = f'(i,j) \stackrel{\text{def}}{=} [i(\text{mod } l), i+j]$, which is not one-to-one, and then (2) shuffling the processes that are mapped to the same invocation $[s,t]$. The shuffling can be performed in many different ways, as long as it respects "$\prec$", the data dependency relation on the DAG. Note that in Crystal's parallelism, load balancing during execution does not become a problem for programs having data independent flows (such as this example). In contrast to the case of a program interpreted by applicative parallelism where mappings of computation to smaller networks create exponential congestion in some of the processors, and load balancing becomes a series problem, as shown in Martin's Torus [16].

A similar modification that performs projection followed by shuffling can be applied in the case where the dimensionality of the basis communication vectors exceeds that of the network.

4. Optimizing Compilers

A compiler for a sequential programming language must allocate resources such as registers in an intelligent way. Analogously, a compiler for a parallel programming language must allocate resources, this time processors and communication channels, rather than those resources within a processor. The central concern of a parallel compiler is the trade-off between the computations local to each processor and the communications between processors.

Beyond the optimization performed on the space-time recursion equations mentioned above, an algorithm can be further improved, or optimized at the source level to yield better parallel programs.

4.1. Optimal dimensionality

As illustrated above, a program with a k-dimensional process structure is compiled to a design using a $(k-1)$-dimensional network. If the dimensionality of the actual machine is less than $k-1$, then the design must be projected down to be fitted on the machine. If, on the other hand, the dimensionality of the machine is greater, one might suspect whether or not the power of the machine is fully utilized. The notions of *fixed fan-ins* and *fixed fan-outs*, motivated by the limitation on the bandwidth for communication at each processor, are used to characterize whether or not the process structure of a program is an optimal one. The number of fan-ins of a program is the maximum number of terms appearing on the right-hand side of any of the recursion equations of the program. The number of fan-outs of a program is the maximum of the total number of times a particular datum appears on the right-hand side of all of the recursion equations. In any physical implementation, the bandwidth of communication channels is fixed, not proportional to a problem size, and therefore the growth of fan-ins (or fan-outs) with the problem size must be avoided. When a program does not have fixed fan-ins or fixed fan-outs, the dimensionality of its process structure must be increased to allow a greater degree of connectivity in the hope of bringing the number of fan-ins and fan-outs down to constants. An illustration of such optimization (stage 1 of the compilation process) by transforming algebraically the definition of a problem to a system of recursion equation that has low fan-in and fan-out degrees and is low order, i.e., using only local communication, is illustrated in [5].

4.2. Detecting common expensive computations

Similar to the idea of detecting common sub-expressions in conventional compilation, detecting *common expensive computation* can reduce the cost of implementation. An expensive computation is one that takes more time than the computation (or communication) time in an otherwise balanced implementation. Hence more powerful computing resources are dedicated to this operation to keep up the performance. The total cost of the computation can be reduced if the results computed by the process using the expensive resource can be transferred to other processes rather than having the common expensive computation be carried out by many processes.

Such an optimization technique can be applied to STREQ at the source level as illustrated by the following example. The program for LU decomposition of a matrix consists of recursion equations for three data streams: stream a for the matrix to be decomposed, and streams l and u containing respectively the lower and upper triangular matrices to be obtained. The system of recursion equations is defined on process structure N^3, where $N \stackrel{\text{def}}{=} \{0,1,2,\ldots,n\}$. The Table 1 contains the part of the program that is transformed to yield a better design by reducing the number of division operations needed from order

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

Table 1: Code optimization for LU decomposition

n^2 to n. The modification consists of moving the processes performing divisions in a region of the half plane $i > k = j$ to those in a segment of the line $i = j = k$ by using an extra data stream b which flows along the same direction as stream u. Stream b can be sent on the same channel as u.

5. Conclusions

5.1. Results

The language of recursion equations provides a simple, straight-forward, and conceptually clean way of describing algorithms. When interpreted by **Crystal**'s parallelism, it naturally discloses the communications between processes and the computation within an individual process. The **Crystal** compiler takes these equations and automatically generates an efficient parallel program. Among the parallel programs, or VLSI architecture generated are: a design for LU decomposition of matrices [6] that is three times more efficient than that in [13], a class of fast multipliers for the long multiplication algorithm [4], and new and efficient parallel implementations for integer partitions, transitive closure, and dynamic programming [7]. Thus based on language **Crystal**, a unified theory for understanding and generating any systolic design comes into existence.

5.2. Language Issues

To the questions raised in the very beginning of this paper, **Crystal** provides its answers: A programmer need not specify explicit communications in a program; the specification is a functional definition that is natural to the problem rather than tailored to a machine. In a **Crystal** program, cost of communications can be extracted from difference vectors, as they are mapped to communication vectors, each of which has a cost associated with the target machine and technology. Similarly, the cost of computation can be extracted from the processing function, which is a composition of a set of base functions, and each has a cost depending on the implementation medium. Thus computation/communication trade-offs can be expressed formally as transformations of **Crystal** programs, and carried out by an optimizing compiler. As discussed in Section 2, recursion equations of the form (2.2) with unbounded minimalization are general purpose, and a process structure can have an arbitrary topology. Last but not least, **Crystal** does encourage large scale parallelism; in fact its phylosophy of computing imposes such parallelism, it allows composition and abstraction of **Crystal** programs in which no implementation details are of any concern.

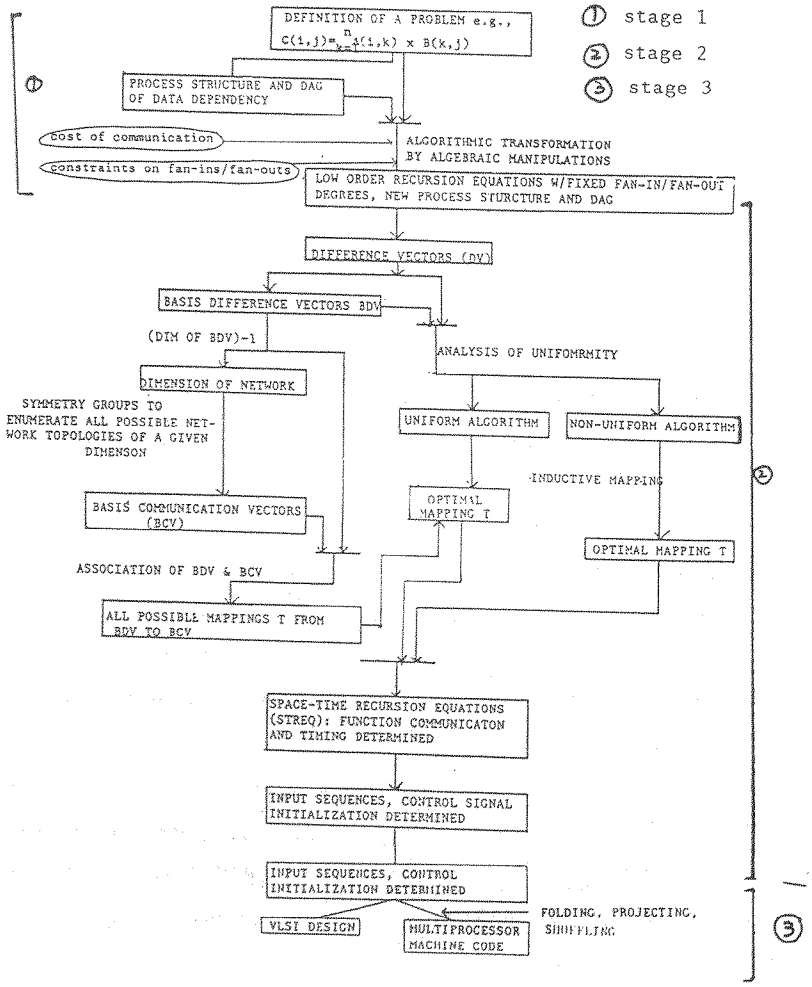

Figure 6: The compilation steps of a **Crystal** program.

Such capability makes **Crystal** appropriate not only as a programming language for problem solving, but also as a system programming language for multi-user operating systems and a specification language for interfacing systems consisting of dissimilar sub-systems. A multi-user operating system can be viewed as a parallel system consisting of loosely coupled user tasks and system resources. The implementation of such an operating system can certainly be described by recursion equations as the interaction of a collection of processes, each being the abstraction of a user program which, at its highest level, consists of a system of recursion equations. Hence the problems of efficient resource allocation at the operating system level becomes very similar to that of **Crystal**'s optimizing compiler, and both can be tackled in a uniform framework.

6. Acknowledgement

I would like to thank Martin Rem for suggesting the integer partition problem as an example for illustrating both the language and the synthesis method. My thanks go to Andrea Pappas and Chris Hatchell for their assistance with the manuscript and, in particular, Andrea for all of her art work.

References

[1] Burstall, R.M., *Proving Properties of Programs by Stuctural Induction,* Computing Journal, 12/1 (1969), pp. 41–48.

[2] Chen M. C., Graham R. L., Rem M., *A Characterization of Deadlock-Free Resource Contention,* Technical Report 4684, California Institute of Technology, January 1982.

[3] Chen, M. C., *Space-time Algorithms: Semantics and Methodology,* Ph.D. Thesis, California Institute of Technology, May 1983.

[4] ————, The Generation of a Class of Multipliers: a Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers,* October 1985.

[5] ————, *Automatic Generation of VLSI Architectures: Synthesis by Algorithm Transformation,* Technical Report 427, Yale University,, October 1985.

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

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

[8] Keller, Robert M. Davis, Alan L., *Data Flow Program Graphs,* IEEE Computer, 15/2 (1982), pp. 26–41.

[9] Devitt, D.J., Finkel, R., Solomon, M., *The Crystal Mlticomputer: Design and Implementation Experience,* Technical Report 553, Department of Computer Science, University of Wisconsin, September 1984.

[10] Gajski, Padua, Kuck, and Kuhn, *A Second Opinion on Data Flow Machines and Languages,* IEEE Computer, 15/2 (1982), pp. 58–69.

[11] Hoare, C.A.R., *Communicating Sequential Processes,* Communication of ACM, 21/8 (1978), pp. 666–677.

[12] INMOS Limited., *OCCAM Programming Manual.,* Prentice Hall, International series in computer science, 1984.

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

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

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

[16] Martin, A.J., The Torus: An Exercise in Constructing a Processing Surface, *Proceedings of Conference on VLSI: Architecture, Design, Fabrication,* 1979, pp. 52–57.

[17] Ousterhout, J.K., Crystal: A Timing Analyzer for nMOS VLSI Circuits, *Third Caltech conference on Very Large Scale Integration,* Computer Science Press, 1983, pp. 71–86.

[18] Scott, D.S. and Strachey, C., Toward a Mathematical Semantics for Computer Languages, Fox, J. ed., *Proceedings of the Symposium on Computers and Automata,* Polytechnic Institute of Brooklyn Press, New York, 1971, pp. 19–46.

[19] Seitz, Charles L., *The Cosmic Cube,* CACM, January (1985), pp. 22–33.

[20] Shapiro, Ehud, *Systolic Programming: A Paradigm of Parallel Processing.,* Technical Report cs84-16, Department of Applied Mathematics, The Weizmann Institute of Science, January 1985.