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 systems 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 program, and there is no concerns of grain-size of parallelism, nor of any implementation technique such as pipelining, for a programmer. The Crystal compiler automatically incorporates pipelining into the program, and generates an implementation that is optimal with respect to the algorithm. An optimizing compiler for Crystal that attempts to balance computations and communications on a target machine is also discussed.

A Parallel Language and Its Compilation
to Multiprocessor Machines or VLSI

Marina C. Chen
Research Report YALEU/DCS/RR-412
August 1985

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. N00014-82-K-0184 and No. N00014-85-K-0030.
Table of Contents

1 Introduction .................................................. 1

2 The Parallel Language Crystal .............................. 2
   2.1 A high level parallel program ...................... 2
   2.2 Applicative parallelism ............................. 2
   2.3 Crystal's parallelism ................................ 2
   2.4 Advantage of Crystal's parallelism ................. 3
   2.5 Recursion equations ................................ 4
   2.6 Correctness of a Crystal program ................. 5
   2.7 Structured programming in Crystal ................. 5
   2.8 Space-time recursion equations ................... 5

3 Compilation Method ........................................ 6
   3.1 Process structure and communication functions ... 6
   3.2 Basis difference vectors ............................ 7
   3.3 Uniformity of a parallel program .................. 7
   3.4 Basis communication vectors ........................ 7
   3.5 Inductive mapping: a constraint optimization problem .. 8
   3.6 Compiling STREQ to machine code or VLSI .......... 9
      3.6.1 Processor and time requirements .............. 9
      3.6.2 Input ports and storage requirements ........ 9
      3.6.3 Control requirements .......................... 10
      3.6.4 Optimization performed on STREQ ............ 10
   3.7 Limited number of processors and network dimensions .. 10

4 Optimization at Algorithmic Level ....................... 10
   4.1 Optimal dimensionality ............................. 11
   4.2 Detecting common expensive computations .......... 11

5 Results and Future Work .................................. 12

6 APPENDIX .................................................. -1

Bibliography .................................................. -4
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. Towards 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[10]), 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 program (as in [6, 12, 7]) and there is no concerns of grain-size of parallelism, nor of any implementation technique such as pipelining, for a programmer. The Crystal compiler finds an parallel implementation of a program and automatically incorporates pipelining, for it to be executed on a multiprocessor machine or to be realized in VLSI. The implementation is optimal with respect to the algorithm, however, its actual performance depends on the relative communication and computation speed of the target machine. Examples of optimization to balance computation and communication are discussed.

Comparing with other programming environments that allow efficient parallel implementations such as Shpiro's systolic programming in concurrent Prolog [14] and Snyder's Poker programming environment, Crystal takes one step further in automating the process of mapping algorithms to correct and efficient parallel implementations. Moreover, an implementation is expressed in terms of recursion equations also, the same language as that for an algorithm. Hence in Crystal, the equivalence of an algorithm and its implementation can be shown formally and guaranteed mechanically.

In Section 2, the language Crystal and its interpretation are introduced. The compilation method of Crystal is given in in Section 3. Techniques for an optimizing compiler are discussed in Section 4. In the last section, a few efficient parallel implementations of Crystal programs are mentioned. Finally, the above language issues are revisited to review the capability of Crystal and the future work that needs to be accomplished.
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 \((m, k)\).

\[
C(i,j) = \begin{cases} 
  i = 1 & \rightarrow 1 \\
  i > 1 & \rightarrow \begin{cases} 
    \text{if } j > i & \rightarrow C(i-1,j) \\
    \text{if } j = i & \rightarrow C(i-1,j) + 1 \\
    \text{if } j < i & \rightarrow C(i-1,j) + C(i,j-i) 
  \end{cases}
\end{cases}
\] (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 \( C(4,7) \).](image)

2.3. Crystal's parallelism

In Crystal, however, a completely different interpretation is given to the equation, and parallelism 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 \( D \triangleq \{(i,j) : 0 < i \leq m, 0 < j \leq k\} \), 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 appear 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 \( v \) (\( u < v \)), or \( v \) depends on \( u \) (\( v \succ u \)), if there is an 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 a top-down view of a computation, 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 a bottom-up one 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.

2.4. Advantage of Crystal's parallelism

Is there any advantage to Crystal's interpretation of recursion equations? The asymptotical complexity for computing \( C(m, k) \) by applicative parallel evaluation amounts to \( O(m + k) \) for the time requirement and \( O\left(\frac{k}{\min(k-1, m)}\right)^m \) 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 leave 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 \( 1 + \frac{k}{2} + \left(\frac{k}{3}\right)^2 + \cdots + \left(\frac{k}{m}\right)^{m-1} \) for \( k > m \), where each term \( \left(\frac{k}{m}\right)^{m-1} \) is the asymptotical 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. 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.

Based on its bottom-up view of parallelism, Crystal's compiler will find optimal mappings of processes (nodes in the DAG) to processors (in a multiprocessor machine) to yield efficient parallel implementation of
a program. Many of the mappings yield programs that are systolic, hence the compilation method can also be viewed as a synthesizer which generates systolic designs from a program specified in a high level language.

2.5. 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

\[
F_1(v) = \phi_1(F_1(\tau_{11}(v)), F_2(\tau_{12}(v)), \ldots, F_n(\tau_{1n}(v)))
\]

\[
F_2(v) = \phi_2(F_1(\tau_{21}(v)), F_2(\tau_{22}(v)), \ldots, F_n(\tau_{2n}(v)))
\]

\[\ldots\]

\[
F_n(v) = \phi_n(F_1(\tau_{n1}(v)), F_2(\tau_{n2}(v)), \ldots, F_n(\tau_{nn}(v))),
\]

where

- \( v \overset{\text{def}}{=} [v_1, v_2, \ldots, v_n] \in A \), \( A \) 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 \((A, \sqsubseteq)\) is a flat lattice, where \( \sqsubseteq \) is the approximation order [13].

- Function \( \tau_{ij} \) 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,

\[
\lambda(F_1, F_2, \ldots, F_n) = \lambda v.[\phi_1(\ldots), \phi_2(\ldots), \ldots, \phi_n(\ldots)]
\]

is a continuous function mapping from \( E^n \) to \( E^n \), where \( E \overset{\text{def}}{=} [(A, \sqsubseteq) \rightarrow (D, \sqsubseteq)] \), 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 \( \phi_i \), and communication functions \( \tau_{ij} \), where \( 1 \leq i, j \leq n \). 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 where 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 \( \phi_i \) 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

\[
F_i(v) = \begin{cases}
  p_{i1}(F_1(\tau_{11}(v)), F_2(\tau_{12}(v)), \ldots, F_n(\tau_{1n}(v))) \rightarrow \phi_{i1}(F_1(\tau_{11}(v)), F_2(\tau_{12}(v)), \ldots, F_n(\tau_{1n}(v)))
  \\
  p_{i2}(F_1(\tau_{21}(v)), F_2(\tau_{22}(v)), \ldots, F_n(\tau_{2n}(v))) \rightarrow \phi_{i2}(F_1(\tau_{21}(v)), F_2(\tau_{22}(v)), \ldots, F_n(\tau_{2n}(v)))
  \\
  \ldots
  \\
  p_{ik}(F_1(\tau_{k1}(v)), F_2(\tau_{k2}(v)), \ldots, F_n(\tau_{kn}(v))) \rightarrow \phi_{ik}(F_1(\tau_{k1}(v)), F_2(\tau_{k2}(v)), \ldots, F_n(\tau_{kn}(v)))
\end{cases}
\]

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, corresponding to the so called single instruction multiple data (SIMD) type of machine architectures, the processing and communication functions of all elements of the process structure are the same, and 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.

2.6. 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.

2.7. Structured programming in Crystal

Similar 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 systems 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 independent 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.

2.8. Space-time recursion equations

From an implementation point of view, a process \( v \) in an 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 a 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 \( v \) 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 substitute 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 a 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) [2] 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 program, 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 3.

![Compilation Diagram]

**Figure 3**: The compilation steps of a Crystal program.

In the following, the compilation method is illustrated by deriving a design to solve the 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 \( \tau_2(i,j) = (i, j - i) \). Note that for each different \( i \), there is associated a different communication function \( \tau_2 \). This variation suggests that the partition problem may need to be solved by a design that is not quite regular.
3.2. Basis difference vectors

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 \(\tau_2(i, j)\), which appear on the right hand side of the equation. The difference vectors obtained from this equation are \(\hat{\tau}_1 \overset{\text{def}}{=} (1, 0)\), and \(\hat{\tau}_i \overset{\text{def}}{=} (0, i)\) for \(0 < i \leq m\).

3.3. Uniformity of a parallel program

The concept of uniformity is introduced to characterize parallel programs so that an expedient procedure can be applied to a uniform program 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 coefficients.

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 \(\Delta t\), and which represents a communication from a processor \(s\) to another processor \(s + \Delta s\) in \(\Delta t\) 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 coefficient 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 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 becomes non-linear. An inductive mapping procedure described below will be used to find the space-time mapping for non-uniform programs.

The program (Equation 2.1) for the partition problem is non-uniform, as for each \(i\), a different set of basis vectors \(\{\hat{i}, \hat{j}\}\) is needed to express the difference vectors appearing in Equation (2.1).

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 three dimensional network as an example, the restriction of nearest neighbor connection can be relaxed to allow, for instance, a diagonal connection whose communication vector is \([1, 1, 0, 1]\). Since in general there can be more than two processors along each dimension of a network, the communication vector \([1, 0, 0, 1]\) is different from but as valid to be a basis as \([-1, 0, 0, 1]\). 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 machine 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 \( G \) defined as \( \{(E, r), R_{180}\) 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 \([9]\) can be generalized to an \( n \) dimensional network. As shown in Figure 4, each subgroup is identified with a set of basis communication vectors that describes the data flow at each processor.

![Symmetry groups and the sets of communication vectors](image)

**Figure 4:** Symmetry groups and the sets of communication vectors that describes the data flow

The sets of communication vectors, each of which corresponds to a subgroup of \( G \), consist of \( \{\vec{r} \in [1, 1], \hat{r} \in [-1, 1]\} \) (\( G \) itself as the subgroup), and \( \{\hat{r} \in [0, 1], \hat{r} \in [0, 1]\} \) (subgroup \( \{E\} \)). For each \( i, 0 < i \leq m \), all possible mappings from a set of basis difference vectors to a set of basis communication vectors are:

\[
T_{1i} : \begin{pmatrix} \hat{r} \\ \hat{j} \end{pmatrix} \mapsto \begin{pmatrix} \hat{r} \\ \hat{j} \end{pmatrix} \quad T_{2i} : \begin{pmatrix} \hat{r} \\ \hat{j} \end{pmatrix} \mapsto \begin{pmatrix} \hat{r} \\ \hat{j} \end{pmatrix} \quad T_{3i} : \begin{pmatrix} \hat{r} \\ \hat{j} \end{pmatrix} \mapsto \begin{pmatrix} \hat{r} \\ \hat{j} \end{pmatrix}
\]

i.e. \( T_{1i} = \begin{pmatrix} 1 & 1 \\ 0 & i \end{pmatrix} \quad T_{2i} = \begin{pmatrix} 0 & 1 \\ 1/i & 1 \end{pmatrix} \quad T_{3i} = \begin{pmatrix} 1 & 1 \\ -1/i & 1 \end{pmatrix} \)

If the program were uniform, then the mapping between the two basis sets would have given the linear mapping from \((i, j)\) to a space-time index pair \((s, t)\) immediately. However, the program is non-uniform, and hence the mapping non-linear, the inductive mapping procedure must be invoked. The design using mapping \( T_{1i} \) for the two set of basis vectors is derived below, others can be found in the Appendix.

3.5. Inductive mapping: a constraint optimization problem

To find the space-time mapping \( f : (i, j) \mapsto (s, t) \), we proceed inductively on the DAG (which is a well-founded set). The idea is that we use the knowledge about the mapping \( f \) up to some nodes in the DAG, and find the mapping for nodes that depend on them. To obtain the mapping \( f(i, j) \) of a node \((i, j)\) assuming the mappings of nodes \((i-1, j)\) and \((i, j-1)\) are known, we need to solve a constraint optimization problem that minimizes the maximum vector length of the vectors \( \hat{T}_{1i}R_1 \) and \( \hat{T}_{1i}R_2 \) and satisfies the synchronization equality

\[
\begin{align*}
[s, t] &= f(i, j) := f(i - 1, j) + \hat{T}_{1i}R_1 \\
&= f(i, j - i) + \hat{T}_{1i}R_2 \\
\text{where } R_1 &\overset{\text{def}}{=} \begin{pmatrix} p_1 & 0 \\ 0 & p_1 + q_1 \end{pmatrix}, R_2 &\overset{\text{def}}{=} \begin{pmatrix} p_2 & 0 \\ 0 & p_2 + q_2 \end{pmatrix} \text{ are routing matrices} \\
\text{and } p_l > 0, q_l \geq 0 \text{ for } l = 1, 2.
\end{align*}
\]

(3.1)
The routing matrices chosen here require that for each basis difference vector \( \vec{i} \) or \( \vec{j} \), movement along the corresponding basis communication vectors (recall that \( \vec{r} = \vec{i} T_{11} \) and \( \vec{r} = \vec{j} T_{11} \)) be made at least once (\( p_{11} > 0 \)). On the other hand, \( q_{i} \geq 0 \) is a condition which allows extra delays be introduced at each communication for the purpose of synchronization. The details in solving this optimization problem is included the Appendix. The resulting optimal space-time mapping is the following.

**Proposition 3.1.** The optimal mapping \( f \) is defined by \( f = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \) such that \( [s, t] = f(i, j) = [i, i + j - 1] \) for \( 0 < i \leq m \) and \( 0 < j \leq k \).

Since \( f \) is a 1 to 1 function, its inverse \( f^{-1} \) maps each invocation \([s, t]\) to a process \((i, j)\). Knowing the inverse space-time mapping \( f^{-1} \), the STREQ for the partition problem can be obtained. Let \( Q \) denote the data stream of the STREQ, i.e., \( Q[s, t] \overset{\text{def}}{=} C(i, j) \), then \( Q = C f^{-1} \).

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

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

(3.2)

where \( 0 < s \leq m \) and \( 0 < t \leq k + m - 1 \)

### 3.6. Compiling STREQ to machine code or VLSI

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

#### 3.6.1. Processor and time requirements

The design consists of 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 5.

![Figure 5: A systolic implementation for the partition problem](image)

### 3.6.2. Input ports and storage requirements

In the STREQ, if a communication vector has 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 \( s \) (\( 0 < s \leq m \))
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.

### 3.6.3. 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 \( t > 2s - 1 \) is satisfied. In an VLSI implementation, however, it is unreasonable to store these two integers of length \( \log(k + m) \) and perform arithmetic operations on them while each processor is only an one-bit processing unit. The following optimization of STREQ is needed.

### 3.6.4. Optimization performed on STREQ

A better design can be obtained by replacing the expensive computations of the guards by transferring a one-bit control signal. The optimized STREQ derived from Equation (3.2) is the following, where \( R[s, t] \) is a data stream that carries the one-bit control signal.

\[
Q[s, t] = \begin{cases} 
    s = 1 & \rightarrow 1 \\
    s > 1 & \rightarrow \begin{cases} 
        R[s, t] = 1 & \rightarrow Q[s - 1, t - 1] + 1 \\
        R[s, t] = 0 & \rightarrow Q[s - 1, t - 1] + Q(s, t - s)
    \end{cases}
\end{cases}
\]

\[
R[s, t] = \begin{cases} 
    s = 1 & \rightarrow \begin{cases} 
        t = 1 & \rightarrow 1 \\
        t > 1 & \rightarrow 0
    \end{cases}
\end{cases}
\]

Central to an optimizing compiler for parallel programs is making trade-offs between communications and computations, as illustrated here and in Section 4 below.

### 3.7. 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:

\[
[s, t] = f^*(i, j) \overset{\text{def}}{=} [i \mod l, i + j - 1 + \lfloor j/l \rfloor \times l], \text{ where } "/\text{" is the integer division operation.}
\]

This mapping based on \( f \) is obtained by (1) first "folding around" the network of size \( l \) to obtain a function \( [s, t] = f'(i, j) \overset{\text{def}}{=} [i \mod l, i + j - 1], \) which is not one-to-one, and then (2) shuffling the processes that are mapped to the same alias \( [s, t] \). The shuffling can be performed in many different ways, as long as the resulting function \( f^* \) is monotonic with respect to "\(<\"", 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 when a program interpreted by applicative parallelism, mappings of computation to smaller networks creates exponential congestion in some of the processors, and load balancing becomes a series problem in Martin's Torus [11].

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. Optimization at Algorithmic Level

Beyond the optimization techniques above, a program can be further improved, or optimized at the source level to yield better parallel implementations.
4.1. Optimal dimensionality

As illustrated above, a program with an $n$ dimensional process structure is compiled to a design using an $n-1$ dimensional network. If the dimensionality of the actual machine is less than $n-1$, 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-out, 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 a problem size, 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-out, 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 example of such optimization of programs by algebraic manipulation 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 result 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 $N \triangleq \{0, 1, 2, \ldots, n\}$. The following table contains the part of the program that is transformed to yield a better design by reducing the number of division operations needed from order $n^2$ 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 $u$.

<table>
<thead>
<tr>
<th>left hand side</th>
<th>conditions</th>
<th>right hand side</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l(i,j,k)$</td>
<td>$(j = k) \land (i &gt; k)$</td>
<td>$a(i,j,k - 1) \times [u(i,j,k)]^{-1}$</td>
</tr>
<tr>
<td>$u(i,j,k)$</td>
<td>$(j = k) \land (i = k)$</td>
<td>$a(i,j,k - 1)$</td>
</tr>
<tr>
<td></td>
<td>$(j = k) \land (i &gt; k)$</td>
<td>$u(i - 1,j,k)$</td>
</tr>
<tr>
<td>$b(i,j,k)$</td>
<td>$(j = k) \land (i = k)$</td>
<td>$a(i,j,k - 1))^{-1}$</td>
</tr>
</tbody>
</table>
5. Results and Future Work

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 implementation. Among the parallel implementations generated are: an design for LU decomposition of matrices [5] that is three times more efficient than that in [8], a class of fast multipliers for the long multiplication algorithm [3], and new and efficient parallel implementations for integer partitions, transitive closure, and dynamic programming [4].

Quite a few interesting questions must be answered before Crystal has a powerful optimizing compiler, to name a few of them:

- What are the general methods for finding the optimal dimensionality of a program's process structure?
- How should load balancing be handled for programs having data dependent flows?
- Given parameters such as the relative speed of communication and computation, the size and dimensionality of the network, what is the best way to balance communication and computation in a program? What is the optimal grain size for parallelism? what is the optimal size of a sub-network for a particular program in a multi-user environment?

Any of these issues involves the relationship between communications and computations. It is encouraging that in Crystal, optimization techniques such as detecting common expensive computations, also involved trade-offs between communication and computation, can be expressed quite eloquently.

To the questions raised in the very beginning of this paper, Crystal provides its answers: A programmer needs not to 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, 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.
6. APPENDIX

The inductive mapping

I. The base case: \( i = 1 \). From Equation (2.1), \( C(1, j) \) is defined unconditionally. Hence the index pairs \((1, j)\) for \( 0 < j \leq k \) can be mapped by the linear transform \( T_1 \) without any constraint, i.e., using only the linear transformation

\[
 f(1, j) = (1, j)T_1 + a\hat{r} + b\hat{t} = (1, j) \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} + a\hat{r} + b\hat{t}
\]

where \( a, b \) are integer constants.

We choose \( a = 0 \), and \( b = -1 \), and thus \( f(1, j) = [1, j] \).

II. Inductive step \( i > 1 \) and \( j \leq i \). In this case, the synchronization equality (3.1) reduces to

\[
 f(i, j) := f(i - 1, j) + [p_1, p_1 + q_1],
\]

thus the optimization problem is not subject to any constraint. It is obvious then that the optimal solution for \( f(i, j) \) is obtained when \( p_1 = 1 \) and \( q_1 = 0 \), i.e., \( f(i, j) = f(i - 1, j) + [1, 1] \)

III. Inductive step \( i > 1 \) and \( j > i \). The synchronization equality becomes

\[
 (s, t) = f(i, j) := f(i - 1, j) + [p_1, p_1 + q_1] = f(i, j - i) + [0, p_2 + q_2].
\]  

(6.1)

Let \( f_1(i, j) \) and \( f_2(i, j) \) be the first and second component of \( f(i, j) \). Note that \( f_1(i, j) \) is uniquely determined by the synchronization equality. If there were no positive integer of \( p_1 \) that satisfied Equality (6.1), then there would exist no design for mapping \( T_1 \). For this problem, the following proposition holds.

**Proposition 6.1.** To satisfy the synchronization equality, \( p_1 \) must be set to 1, i.e., \( f_1(i, j) = f_1(i, j - i) \) for \( 0 < i \leq m \) and \( i < j \leq k \), and \( f_1(i, j) = f_1(i - 1, j) + 1 \) for \( 1 < i \leq m \) and \( 0 < j \leq k \).

**Proof.** By induction on the DAG. In part I above, we have \( f_1(1, j) = f_1(1, j - 1) \) for all \( 1 < j \leq k \). By part II above, we have \( f_1(i, j) = f_1(i - 1, j) + 1 \) for \( 1 < i \leq m \) and \( j \leq i \). Then

\[
 f_1(i, j) = f_1(i, j - i) \text{ by the synch. eq.}
 = f_1(i - 1, j - i) + 1 \text{ by ind. hyp.}
 = f_1(i - 1, j) + 1 \text{ by ind. hyp.}
 \Rightarrow p_1 = 1 \text{ by the synch. eq.}
\]

What is left to be determined in Equality (6.1) is a positive integer \( r_1 \geq p_1 = 1 \) and a positive integer \( r_2 = p_2 + q_2 \). Since the first component \( f_1(i, j) \) is determined, the feasible space for \( f(i, j) \) is a ray parallel to the \( t \) axis starting from the point

\[
 z \overset{\text{def}}{=} (f_1(i, j), \max (f_2(i - 1, j) + 1, f_2(i, j - i) + 1))
\]  

(6.2)

going in the positive direction, as shown in Figure 6. The objective function for optimization is to minimize the maximum vector length of \( [1, r_1] \) and \( [0, r_2] \); it assumes minimum value at \( x \), i.e., \( f(i, j) = x \). From the following proposition, \( r_1 \) and \( r_2 \) can be determined.
Proposition 6.2. (a) $f_2(i, j) > f_2(i, j - 1)$ for $1 < j \leq k$, (b) $f_2(i - 1, j) > f_2(i, j - i)$ for $1 < i \leq m$ and $i < j \leq k$, and (c) $f_2(i, j) = f_2(i - 1, j) + 1$ for $1 < i \leq m$ and $0 < j \leq k$, i.e., $r_1 = 1$ and $r_2 = f(i - 1, j) - f(i, j - i) + 1$ for $j > i$.

Proof. By induction on the DAG. The base case for (a) is when $i = 1$, and it is true by part I above. The base case for (b) is when $i = 2$ and $j = 3, 4$. By I above, $f_2(1, j) = j$. By II above, $f_2(2, 1) = f_2(1, 1) + 1 = 2 < f_2(1, 3)$ and $f_2(2, 2) = f_2(1, 2) + 1 = 3 < f_2(1, 4)$. The base case for (c) is when $1 < i \leq m$ and $0 < j \leq i$ and it holds by part II above. For the general case, we assume the induction hypothesis holds for all pairs that precede $(i, j)$ in the DAG. Then (a) holds for $(i, j)$ because

\[
 f_2(i, j) \geq f_2(i - 1, j) + 1 \quad \text{(by Definition of } x \text{ in Equation (6.2))}
\]

\[
 > f_2(i - 1, j - 1) + 1 \quad \text{(by (a) of ind. hyp.)}
\]

\[
 = f_2(i, j - 1) \quad \text{(by (c) of ind. hyp.).}
\]

Also by induction hypothesis, (b) holds because

\[
 f_2(i - 1, j) = f_2(i - 2, j) + 1 \quad \text{(by (c))}
\]

\[
 > f_2(i - 1, j - (i - 1)) + 1 \quad \text{(by (b))}
\]

\[
 > f_2(i - 1, j - i) + 1 \quad \text{(by (a))}
\]

\[
 = f_2(i, j - i) \quad \text{(by (c)).}
\]

The general case for (c) holds as a corollary of (b) and the definition of $x$ (Equation (6.2)).

By Proposition 6.1 and part (c) of Proposition 6.2, we have

\[
 f(i, j) = \begin{cases} 
 i = 1 & \rightarrow [1, j] \\
 i > 1 & \rightarrow f(i - 1, j) + [1, 1].
\end{cases}
\]

Solving this very simple recursion equation, we obtain Proposition 3.1.

![Figure 6: The feasible space for $f(i, j)$](image)

Other designs for solving the partition problem

The design using mappings $T_{2i}$ can be obtained similarly as above, and the space-time mapping function $f$ is also a linear transform defined as $(s, t) = f(i, j) = [j, i + j - 1]$ 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 $\hat{v}_i = (0, i)$ is mapped to a communication vector $[i, i]$ whose first component is non-zero. Figure 7(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 is of $O(m)$. One way to cut down on the number of connections is to decompose $[i, i]$ into compositions of $\hat{r} = [1, 1]$ 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 bandwidth of each channel for communication be proportional to $m$, still not a good solution in terms of VLSI implementation. Figure 7(a) shows the execution traces of the modified design. The same implementation issues arise if it is to be implemented on a multiprocessor machine.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/figure7.png}
\caption{(a) The systolic program using mappings $T_{3i}$, which needs re-configurable connections (b) A modified systolic design which needs channels of bandwidth $O(m)$}
\end{figure}

In summary, the time complexity of this systolic program is $k + m$, same as the derived from $T_{1i}$, but the latency is $k$. The total number of processors required is $k$, and the total channel length (or channel bandwidth in the modified version) is of $O(m^2)$.

The program derived from mappings $T_{3i}$ has problems similar to the one for $T_{2i}$ since the difference vector $\hat{k}_i$ is also mapped to a communication vector that is not stationary in space.
References


