Yale University
Department of Computer Science

Compiling Parallel Programs by Optimizing Performance

Marina Chen, Young-il Choo, and Jingke Li

YALEU/DCS/TR-633
June, 1988

To appear in Journal of Supercomputing, August 1988.

Compiling Parallel Programs by Optimizing Performance

Marina Chen Young-il Choo Jingke Li

Department of Computer Science
Yale University
New Haven, CT 06520
chen-marina@yale.edu choo@yale.edu li-jingke@yale.edu

April 11, 1988

Abstract

This paper describes how Crystal, a language based on familiar mathematical notation and lambda calculus, addresses the issues of programmability and performance for parallel supercomputers. Some scientific programmers and theoreticians may ask, "What is new about Crystal?", or "How is it different from existing functional languages?" The answer lie in its model of parallel computation and a theory of parallel program optimization, and we examine this in the text to follow. We illustrate the power of our approach with benchmarks of compiled parallel code from Crystal source. The target machines are hypercube multiprocessors with distributed memory, on which it is considered difficult for functional programs to achieve high efficiency.

1 Introduction

Rapid development in large scale parallel machines presents a new challenge in software: How can these machines be put to effective use?, and How can *efficient* programs be written for a machine consisting of hundreds, thousands, or even millions of processors without too much difficulty? Since the purpose of parallel processing is to obtain high performance at a reasonable cost, efficiency and cost-performance are of ultimate concern.

Superficially, the idea of parallel processing is simple: allow many processors to work together in order to solve a particular problem. Thus, the length of time it takes to perform a task will be reduced. All sounds well and good, but in order to use parallel machines effectively, many problems must be overcome. First of all, we need algorithms that have enough *concurrency* to take advantage of large numbers of processors. Second, this concurrency must be *conveyed* in some language that will eventually be *translated* into commands that can direct the operations and cooperations of processors. Third, the cooperation between processors may cause problems: when one processor needs to use another processor's result — it needs to find out whether that result will be ready to be used and where it can be found. The more processors a system has, the more *communications* between processors must take place. The danger is that the processors can spend more time telling each other how far they have gotten with their tasks than solving them.

This processor bureaucracy must be brought under control. Another problem is the so-called *hot-spot* phenomenon where requests of services are concentrated at a few processors while others are waiting to be served. Like any large organization, concentration of workload must be smoothed out in order to achieve a higher level of overall performance. Finally, there is a cost associated with communicating information between processors. The more local the communication, the less costly. The benefit attained by taking advantage of *locality* must then be weighed against the cost and the effort spent in arranging for local communications and cooperations. Such arrangements may not always be made in a cost-effective manner. However, when a problem is amenable to such an arrangement, the performance gain can be significant.

In this paper we describe how Crystal, [5,6] a language based on familiar mathematical notation and lambda calculus, (see, for example, [24]) addresses each of the above issues. Scientific programmers and theoreticians wonder "What is new about Crystal?", or "How is it different from existing functional languages?" The answer lies in its model of parallel computation and a theory of parallel program optimization[8]. In addition, the methodology of writing parallel programs. Crystal is by no means a full blown functional language, rather, it is an experiment in the design of a language and a metalanguage for parallel program optimization. Its optimization facility can also be viewed as a way of supporting the emerging functional language standard [15] on parallel machines — in particular, distributed memory machines.

On the issue of concurrency, Crystal's philosophy is to make it easy for programmers to express concurrency. However, whether an algorithm by nature has a lot of concurrency, is an algorithm designer's responsibility. Thus, the design of algorithms and the management of parallelism are treated as two distinctively separate issues. The mathematical nature of the language does not presuppose sequential implementation as conventional programming languages do. A Crystal program does not have extraneous dependencies in the original problem as is the case with conventional languages. The program faithfully embodies whatever concurrency has existed in the original algorithm. Thus, it allows a programmer to focus on algorithmic issues, leaving the grungy and error-prone aspect of managing parallelism to its compiler and run-time system.

Regarding the suitability of specifications for concurrency, we do not encourage programmers to describe the microscopic behavior of each individual processor. Rather, it is best to describe what is to be done by the processors collectively. This *lack* of specification in the source program about each processor's assignment provides the machine with flexibility in interpreting how a collection of processors will execute the programmer's command.

In regard to language constructs for specifying concurrency, we believe that machine characteristics (that affect the design of algorithms) and code optimizations (that improve the target code performance) have mathematical representations that can be reasoned without implementation details. Hence, commands dictating the implementations of interprocessor cooperations — such as "locks" for secure sharing of common data or "send" and "receive" for transmitting messages between processors — are not in Crystal's vocabulary. They are used, rather, in the target code into which a source program is translated.

Crystal's view of computation is that of a *global* data space with non-uniform *communication metrics.* Each data subspace, called a *data field*, has its own shape and topology described as graphs. Code optimizations are specified as *morphisms* between data fields and are carried out as source to

source transformations. If so desired, a programmer can explicitly optimize the target code using data field morphisms. Again, this process can be carried out at the source level with the benefit of the mathematical apparatus. Implicitly by the compiler, or explicitly by a programmer, all optimizations are performed at the source level. Since the "target" program embodies information of the exact microscopic behavior of each individual processor, its translation to any of the many languages on parallel machines becomes a straightforward process.

The richness of the language in expressing shapes and topologies of data fields and the capabilities of abstracting away from implementation details imply that source programs can be ported to different parallel architectures. Whether the target machine has shared-memory or distributed memory, is fine-grained or coarse grained, is small scale or massively parallel, is not an issue for programmers. Different architectures simply induce different communication metrics for compiler optimizations.

The heart of Crystal's compilation process lies in morphisms for keeping communication overhead low (space-time realization, partition morphism), avoiding hot spots (fan reduction), ensuring a balanced load (distributing morphism), and attempting locality whenever possible (contraction morphism). The theory of data field morphisms states that all information for an optimization is embodied by a data field morphism, which is a pair of functions. These functions are in "closed form" for compile-time optimizations, and are of "recursive form" for run-time optimizations. Once a morphism is given, the metalanguage processor is responsible for either generating an optimized program or augmenting the original program with additional code which creates an environment for executing the morphism at run-time. To determine the need for optimization in a source program, the compiler must pick up syntactic cues indicating such a need. To determine what optimization is to be applied, optimization procedures or heuristics are needed. Often, it appears that there are multiple choices of optimization strategies, and an analysis and comparison of the performance models implicated by these strategies becomes necessary.

The rest of this paper is organized as follows: We present an overview of the Crystal system in Section 2. In Section 3, we give a brief introduction to the language and its metalanguage. Next, we present a model of parallel computation with communication metrics in Section 4. Notions of data fields, computational fields, communication metrics, and morphisms are introduced. In Section 5, we describe the morphisms needed for generating efficient target code from an initial mathematical description. In Section 6, we present some of the performance models used in the compiler optimization process. We also present performance results of parallel C code which have been compiled from Crystal on two target machines, namely, the NCUBE and the Intel iPSC. We conclude with a discussion of related work and a few remarks.

2 Crystal System Overview

In the following, we outline the current Crystal system which is under design and implementation. Optimizing or improving programs for parallel execution is at the heart of the implementation of Crystal. In Crystal, code optimization may be performed by source-to-source transformations at compile time, or it may be carried out as part of the run-time environment's function. Figures 1 and 2 depict the organization of Crystal compilers for various parallel machines and Crystal's

run-time environment, respectively.

2.1 Optimization Library

Procedures or heuristics for optimizations are collected in the *Optimization Library*, whose functions include: analyzing a source program, choosing what type of optimizations must be applied, specifying the transformation steps for compile-time optimizations, and determining the run-time optimization and activation structure.

2.2 Metalanguage Processor

Compile-time transformations are specified in Crystal's metalanguage, which contains operators for manipulating source programs. The metalanguage processor knows about the algebra of Crystal programs and actually carries out the program transformations. While the metalanguage is used mostly by the procedures in the optimization library, users of Crystal may program in it for explicitly controlling transformations.

2.3 Run-time Environment

Run-time optimizations are achieved by a data/task distributor and a dynamic processor/storage allocator. The former is responsible for shuffling data around for the purpose of load balacing and minimizing communication overhead. The latter is responsible for creating space for procedure activations, establishing communications that transfer data between parallel procedures, and optimizing spatially the location of an "activation record", again, for the purpose of balanced load and low communiation overhead. The garbage collector and storage compactor work across processors as well as on the memory within a processor. The run-time environment itself is implemented in parallel code for minimizing run-time overhead.

2.4 Code Generators

Crystal is an architecture-independent language, which is currently targeted to a variety of parallel machines: hypercube multiprocessors (NCUBE and Intel's iPSC), multiprocessors with mesh interconnections (Ametek's Ginzu), shared-memory multiprocessor (Encore's Multimax), systolic array processors (Warp), and massively parallel machines (Thinking Machine's CM-1 and CM-2). Different architectures impose different optimization criteria and choices of parameters, which are reflected in the optimization procedures in the library.

For most of the machines, Crystal's code generators produce target code consisting of programs for individual processors in high-level languages such as C, Fortran, or Lisp, plus communication or synchronization commands for inter-processor communications. Two exceptions are in the use of *Lisp code along with calls to PARIS as target code for the Connection Machines, and W2 code for the Warp array processor.

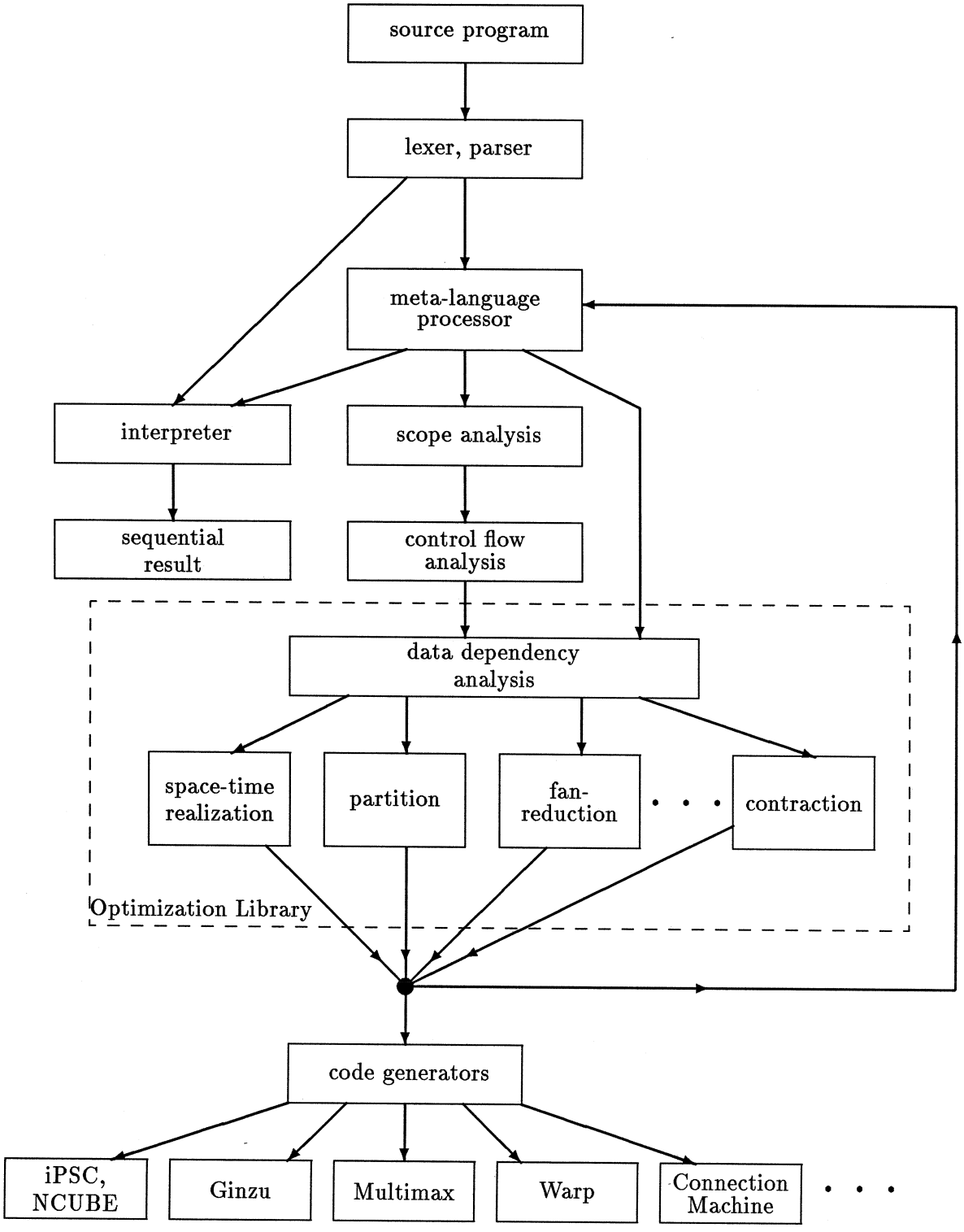

Figure 1: Organization of the Crystal Compilers

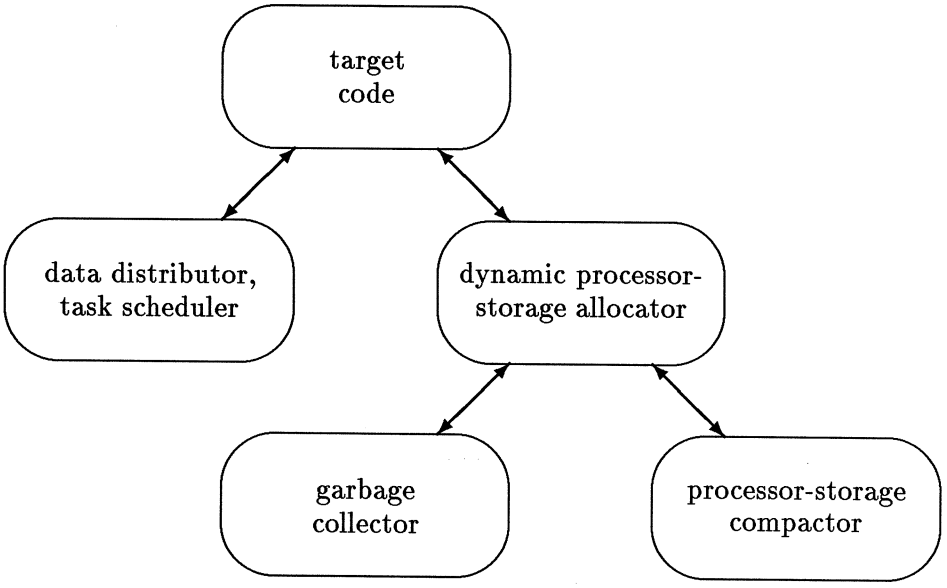

Figure 2: Run-time Environment of Crystal

3 The Crystal Language

The semantics of Crystal includes index domains, domain morphisms, data fields, and computation fields. The syntax of Crystal is basically the mathematical notations used in defining the various mathematical objects, the λ-abstraction and application from the λ-calculus, enriched with recursion and environments. The only control structure consists of the conditional expression. Conventional control structures such as various forms of loops are subsumed by Crystal's domain operators.

3.1 The Language

Data Types

The *basic data types* consist of integers and Booleans with the standard arithmetic functions (plus, minus, times, divide, etc.), and Boolean functions (and, or, not). The standard environment has names for all the integer and Boolean constants, and the standard functions over them.

The *composite data types* include sets, index domains, and data fields. A simple data field a over an interval domain $0 \,..\, n$ can be expressed $[a(0), \ldots, a(n)]$, or $[a(i) \mid i : 0 \,..\, n]$ using data field comprehension. In general, for any domain D, the data field $a : D \to V$ can be expressed as $[a(x) \mid x : D]$. Here $i : D$ indicates that the variable i ranges over the domain D.

The Conditional

The *conditional expression* has the following form:

$$\left\{ \begin{array}{c} B_1 \rightarrow E_1 \\ \vdots \\ B_n \rightarrow E_n \end{array} \right\}$$

where the B_i's are Boolean expressions and E_i's are any expressions. Its value is the value of the first expression with a true guard.

Functions

Given any expression in the language, the λ-*abstraction* produces λ-expressions that denote functions. If the formal parameters are declared over an index domain it denotes a data field.

Example If $e[x]$ is an expression in x, then

$$\lambda x : D.e[x]$$

denotes a data field over D whose values at each index x is $e[x]$.

Repeated λ-abstraction produces higher-order functions.

Operators

Operators are higher order functions that take other functions as arguments. The standard environment contains the *composition* ($\circ$).

The *reduction* operator ([16]) comes in three flavors: the left associative ($\backslash_L$), the right associative ($\backslash_R$), and the binary-tree associative ($\backslash_B$). The left associative $\backslash_L$ takes a binary associative function f and a linear data field $[a_0, \ldots, a_n]$ and is defined as

$$\backslash_L f\,[a_0, \ldots, a_n] = f(\cdots(f(f(a_0, a_1), a_2))\cdots).$$

The others differ only in the association of the binary function f.

The *scan* ($\backslash\backslash$) operation ([16]) is defined similarly

$$\begin{aligned} &\backslash\backslash f[a_0, \ldots, a_n] = \\ &\quad [a_0, f(a_0, a_1), \ldots, f(\cdots(f(f(a_0, a_1), a_2))\cdots)] \end{aligned}$$

and returns a data field of the same shape with all the partial reductions as values.

Programs

A *definition* has the form $\ulcorner f = E \urcorner$ or $\ulcorner f = E$ **where** $N \urcorner$, where f is an identifier, E is an expression, and N is an environment. An *environment* is a set of mutually recursive definitions. An environment following the "where" in a definition is said to be *local* to that definition and augments the environment in which the definition is evaluated. Of course, the definitions in a local environment may also have their own local environments.

A Crystal *program* consists of a set of mutually recursive definitions and an expression that is to be evaluated in the standard environment.

3.2 A Metalanguage

The metalanguage will be used to formalize Crystal program transformations. For a more detailed exposition see [8]. The metalanguage consists of basic constructors and selectors for each of the constructs in Crystal and operations that manipulate Crystal programs as objects. For a more general account of metalanguages see [23,12].

Constructors, Selectors, and Predicates

For each construct of Crystal, the metalanguage provides constructors that take the components and produces the construct. The selectors pick out the specific component of each construct, and predicates tests the type of a component.

Example

$$\begin{aligned} \textsf{mk-eqn}(\tau_1, \tau_2) &\equiv \ulcorner \tau_1 = \tau_2 \urcorner \\ \textsf{lhs}(\textsf{mk-eqn}(\tau_1, \tau_2)) &= \tau_1 \\ \textsf{rhs}(\textsf{mk-eqn}(\tau_1, \tau_2)) &= \tau_2 \end{aligned}$$

$$\begin{aligned} \textsf{mk-abs}(\pi, \tau) &\equiv \ulcorner \lambda \pi . \tau \urcorner \\ \textsf{param}(\textsf{mk-abs}(\pi, \tau)) &= \pi \\ \textsf{body}(\textsf{mk-abs}(\pi, \tau)) &= \tau \end{aligned}$$

are the constructors and selectors for making equations and λ-abstraction.

Operators

The metalanguage operators modify Crystal programs. For example, the standard program transformation"unfold" [4,10] takes a function call and replaces it with the body of its definition, and "fold" does the reverse. These can be defined in terms of more primitive operators: "expand" which replaces the function name by its defining equation, and "reduce" which does β-reduction on a redex.

In the metalanguage, these are denoted $\mathsf{fold}(\kappa, \phi)$, $\mathsf{unfold}(\kappa, \phi)$, $\mathsf{expand}(\kappa, \phi)$, and $\mathsf{reduce}(\tau)$, where κ denotes an equation, ϕ a name, and τ a term.

Other operators include substitution, denoted $\mathsf{subst}(\kappa, \tau_1, \tau_2)$ (in equation κ substitute sub-term τ_1 with τ_2) and various simplifications for arithmetic expressions or function compositions.

The reshape operator, denoted $\mathsf{reshape}(\mathsf{def}(\ulcorner a \urcorner), \ulcorner g \urcorner, \ulcorner b \urcorner)$, is a composite function in the metalanguage. It takes an equation defining the function a and returns an equation defining the function b such that $a = b \circ g$ where g is an index domain morphism.

The fan-in/fan-out reductions can also be expressed as metalanguage functions:

$$\mathsf{fan\text{-}out\text{-}red}(\mathsf{def}(\ulcorner c \urcorner), \ulcorner X_L \urcorner, \ulcorner a \urcorner) \quad \text{and} \quad \mathsf{fan\text{-}in\text{-}red}(\mathsf{def}(\ulcorner c \urcorner), \ulcorner Z_L \urcorner, \ulcorner a \urcorner).$$

The use of these will be illustrated below. A more complete presentation of the metalanguage is in [8].

Fan-in Reduction Consider the program

$$c = \lambda l : D. \backslash f\, H(l),$$

where H has shape $[D \to [N \to V]]$ for some domains D and N and f is some binary associative function. The hot spot occurs at each index l in D since all the values $\{ H(l)(n) \mid n : N \}$ are needed for each l. The fan-in reduction is done by replacing the expression $\ulcorner \backslash f\, H(l) \urcorner$ with $\ulcorner a(l)(r) \urcorner$, where a is a new function whose definition will be added and r is the root of the domain of each $a(l)$.

For the three flavors of reduction, $\backslash_L$, $\backslash_R$, and $\backslash_B$, the definition of a will be different. For the left-associative reduction, the program for c becomes

$$c \;=\; \lambda l : D. a(l)(r) \;\mathsf{where}\; a = Z_L[a, D, H, N, r, f]$$

where

$$Z_L[a, D, H, N, r, f] \;\equiv\; \lambda l : D. \lambda k : \mathrm{tree}_L(N, r). \begin{cases} \mathrm{leaf}(k) \to H(l)(k) \\ \neg \mathrm{leaf}(k) \to f(a(l)(\mathrm{left}(k)), a(l)(\mathrm{right}(k))) \end{cases}$$

and we assume that $a(l)(\perp) = \mathrm{id}(f)$, to take care of the cases when a node k may not have both children. This operation will be denoted $\mathsf{fan\text{-}in\text{-}red}(\mathsf{def}(\ulcorner c \urcorner), \ulcorner Z_L \urcorner, \ulcorner a \urcorner)$.

Similarly, there are corresponding Z_R and Z_B for right associative and binary-tree associative reductions. The only difference comes in the generation of the tree domains tree_R and tree_B.

Tree Fan-Out Reduction Consider the program

$$c = \lambda l : D. e[H]$$

where $e[H]$ is an expression containing H. The interpretation of this program requires that the value H be broadcast to each l in D. To reduce this fan-out, we replace it with a data field a defined over $\mathrm{tree}_B(D, r)$ such that each node of the tree has the value H.

The fan-out reduced program looks like

$$c \;=\; \lambda l : D.e[a(l)] \text{ where } a = X_L[a, D, H]$$

where

$$X_L[a, D, H] \;\equiv\; \lambda l : \text{tree}_B(D, r).\left\{\begin{array}{l} \text{root}(l) \rightarrow H \\ \neg\text{root}(l) \rightarrow a(\text{parent}(l)) \end{array}\right\}$$

This operation will be denoted **tree-fan-out-red**(**def**($\ulcorner c \urcorner$), $\ulcorner X_L \urcorner$, $\ulcorner a \urcorner$).

Butterfly Fan-Out Reduction A more efficient method of fan-out reduction requires a hypercube (Boolean n-cube) topology.

Consider the program

$$a = \lambda j : D.\lambda k : E.V(j)$$

where $D = E = 0 \,..\, n-1$ for some n. At each index k over E, we need the value $V(j)$, which is non-local communication. Using a natural isomorphism of D and a hypercube containing n nodes (assuming for simplicity and without loss of generality that n is some power of 2), we take $\log n$ time steps to copy $V(j)$ to each k.

First, define a syntactic abstraction

$$Q[D, f, V] \;\equiv\; \lambda j : D.\lambda k : 0 \,..\, \dim(D).\left\{\begin{array}{l} k = 0 \rightarrow [V(j)] \\ k > 0 \rightarrow f(j)(k-1) :: f(\text{hcnb}(j,k))(k-1) \end{array}\right\}$$

where $\text{hcnb}(j, k)$ is the neighbor node of j along the kth dimension when D is considered a hypercube, :: is the array append operator, and $\dim(D)$ is the number of dimensions of D considered as a hypercube. For example, a hypercube containing n nodes, the dimension is $\log n$.

Then, given the definition of a, **butterfly-red**(**def**($\ulcorner a \urcorner$), $\ulcorner Q \urcorner$, $\ulcorner \bar{a} \urcorner$) returns the following program

$$\bar{a} \;=\; \lambda j : D.\lambda k : E.f(j)(\dim(D))(k) \text{ where } \left\{ f = Q[D, f, V] \right\}$$

4 A Model of Parallel Programs

In this section, we introduce a model of parallel programs that forms the foundation of an equational theory of program optimizations.

We begin with *index domains* which represent the space of processors with an interconnection topology. Next, *index domain morphisms* between domains formalize the reshaping of index domains. The meaning of a Crystal function defined over a domain is a *data field*, a function assigning values to the elements of a domain. The meaning of a parallel program, presented as a system of mutually recursive definitions, is a *computation field*, a set of interdependent data fields.

The domain morphisms induce *morphisms* between the data fields. The algebra of morphisms of the data fields provide us with an equational theory in which new types of transformations can be defined for improving the overall efficiency of parallel programs.

Index domains are given a topology by defining a communication metric which represents communication cost between different nodes of the domain. This allows us to model any kind of machine architecture, from single processors to shared memory machines, and distributed memory machines. The communication metric provides us with a measure of a program's efficiency on a particular architecture and guides the strategies for parallel-program optimization.

Throughout this section, we shall use V to denote some domain of values. These will typically be the integers, and sometimes higher-order values.

4.1 Graphs and Communication Metrics

By a *graph* we mean a *directed graph*, G, presented as a set of nodes, $\sigma(G)$, and a set of arcs, $\rho(G)$. An arc with source node x and target node y will be denoted $x \mapsto y$. A *path* is a non-empty sequence of arcs such that the target node of one arc is the source node of the next. We write $x \overset{*}{\mapsto} y$ to indicate that there is a path from x to y. A graph has a *finite degree* if each node has a finite number of incoming or outgoing arcs. A graph is *well-founded* if it has no infinite descending chains.

A *graph injection*, written $g : G \to H$, is an injective function from nodes to nodes that preserves paths: if $x \overset{*}{\mapsto} y$ in G, then $\sigma(g)(x) \overset{*}{\mapsto} \sigma(g)(y)$ in H. The *identity morphism* on a graph G, denoted 1_G, is an injection that is the identity on the nodes. A *graph surjection* may collapse arcs with their source and target nodes to single nodes [13]. A *graph morphism* is a composition of graph injections and surjections.

Let G be a graph and R be the non-negative real numbers with infinity.

Definition A *communication metric* on G is a map $\gamma = \gamma(G) : \sigma(G) \times \sigma(G) \to R$, such that

1. If $x = y$, then $\gamma(x, y) = 0$.
2. $\gamma(x, y) \geq 0$ for all x, y.
3. $\gamma(x, z) \leq \gamma(x, y) + \gamma(y, z)$ for all x, y and z.

$\gamma(x, y)$ is called the *communication cost* from x to y.

In Crystal, it will often be convenient to determine a communication metric by specifying the *local metric*, which assigns a cost for each arc. A local metric can easily be extended to be a communication metric.

4.2 Index Domains and Domain Morphisms

An *index domain* D is a graph of finite degree with a communication metric $\gamma(D)$ defined over it. A node of the underlying graph will be called an *index*.

Example A local metric with cost 1 on each arc for a square mesh domain D can be defined as follows:

$$\nu(G) = \lambda x : \rho(G).1 \; .$$

For a shared memory machine, the domain is a star, with each arc having the same communication cost.

Usually, the communication metric will not be given explicitly, but will be inferred from the Crystal program by performing a dependency analysis.

The graph morphisms induce the corresponding *domain morphisms*, with the morphisms acting on the underlying graphs.

An *interval index domain*, denoted $l \,..\, u$, is a domain whose nodes are contiguous integers between l and u, inclusively. The local metric assigns a unit cost to adjacent nodes. We also write $l .< u$ ($l <. u$) if the upper (lower) bound is not included.

A special class of index domains will be used to model discrete time. A *time domain* is a linear, well-founded graph with a default communication metric. In particular, T_n will denote a time domain of length n.

Let D be a domain with n nodes. A *linearization* of D, denoted $\tau(D)$, is a time domain of length n with nodes $\sigma(D)$ such that there exists an injection from D into $\tau(D)$ which is an identity on the nodes. Note that $\tau(D)$ represents just one possible linearization of D.

A *binary tree domain* is a domain with nodes connected as a binary tree. The predicates "root" and "leaf" test for the root and the leaves, and the functions "parent", "left", and "right" return the parent, the left child, and the right child of each node, respectively, if they exist, and is undefined otherwise.

Let S be a set of nodes and r some other node. A *tree domain* over S with root r, denoted $\text{tree}(S, r)$, is a binary tree domain with the leaves from S and the root r. If the cardinality of S is odd, we allow one non-leaf node not to have both children.

A tree domain is *balanced*, denoted tree_B, if the leaves are all at the same distance from the root, is *left-associative*, denoted tree_L, if all the right children are leaves, and is *right-associative*, denoted tree_R, if all the left children are leaves.

Domain Constructions

Starting with simple index domains, new domains can be constructed. One way is to restrict the elements. The usual cartesian product and disjoint union are defined, and a special construction called time product is introduced.

Sometimes we want to restrict ourselves to a subdomain. A subdomain can be defined by providing a *restriction* on a domain, where a restriction is a Boolean valued function over the domain. In Crystal, the restriction is specified using a filter, which is a Boolean expression.

Example

$$\lambda x : 0 \,..\, 100 \wedge even(x).5x$$

denotes a data field defined only over the even members of the domain.

Definition Let D_1 and D_2 be domains. The *cartesian product* of D_1 and D_2, denoted $D_1 \times D_2$,

is defined as follows:

$$\begin{aligned}
&\sigma(D_1 \times D_2) = \sigma(D_1) \times \sigma(D_2) \\
&\rho(D_1 \times D_2) = \\
&\{(x_1, y_1) \mapsto (x_2, y_2) \\
&\quad \mid (x_1 = x_2 \text{ and } y_1 \mapsto y_2) \text{ or } (x_1 \mapsto x_2 \text{ and } y_1 = y_2)\} \\
&\gamma(D_1 \times D_2) = \nu^*
\end{aligned}$$

where the local metric is

$$\nu((x_1, y_1) \mapsto (x_2, y_2)) = \begin{cases} \gamma(D_1)(y_1 \mapsto y_2) & \text{if } x_1 = x_2 \text{ and } y_1 \mapsto y_2, \\ \gamma(D_2)(x_1 \mapsto x_2) & \text{if } x_1 \mapsto x_2 \text{ and } y_1 = y_2. \end{cases}$$

If D_1 and D_2 are interval domains, then the communication metric on their product is usually known as the *Manhattan metric.* Communication can only occur along directions parallel to the axes.

Next, we define the coproduct, or disjoint union.

Definition Let D_1 and D_2 be domains. The *coproduct* of D_1 and D_2 is the disjoint union domain $D_1 + D_2$ with two injections $\iota_1 : D_1 \to D_1 + D_2$ and $\iota_2 : D_2 \to D_1 + D_2$. The communication metric is

$$\begin{aligned}
\gamma(D_1 + D_2)(\iota_1(x), \iota_1(y)) &= \gamma(D_1)(x, y), \\
\gamma(D_1 + D_2)(\iota_2(x), \iota_2(y)) &= \gamma(D_2)(x, y), \text{ and} \\
\gamma(D_1 + D_2)(\iota_1(x), \iota_2(y)) &= k,
\end{aligned}$$

for certain nodes in the coproduct corresponding to x in D_1 and y in D_2 to have communication cost k, called *inter-component cost.*

The injections ι_1 and ι_2 enable us to avoid having to specify the exact implementation of the coproduct in terms of set-theoretic constructions.

When the components of the coproduct are the same, then we define the default inter-component cost for corresponding nodes to be zero, i.e., $\gamma(E + E)(\iota_1(x), \iota_2(x)) = 0$ for all x in E.

Next, we introduce a construction that models a space of processors in time. Unlike the product, where the original arcs remain unchanged, in the following construction, the arcs will point forward in time.

Definition Let S be a domain and T a time domain. The *time product* of S with T, written $S * T$

and read "S in T," is a domain with

$$\begin{aligned}
&\sigma(S * T) = \sigma(S) \times \sigma(T) \\
&\rho(S * T) = \{(x_1, t_1) \mapsto (x_2, t_2) \mid \\
&\quad (x_1 = x_2 \text{ and } t_1 \mapsto t_2) \text{ or } (x_1 \mapsto x_2 \text{ and } t_1 \mapsto t_2)\} \\
&\gamma(S * T) = \nu^*
\end{aligned}$$

where

$$\nu((x_1, t_1) \mapsto (x_2, t_2)) = \begin{cases} \gamma(T)(t_1 \mapsto t_2) \\ \quad \text{if } x_1 = x_2 \text{ and } t_1 \mapsto t_2, \\ \max\{\gamma(S)(x_1 \mapsto x_2), \\ \quad \gamma(T)(t_1 \mapsto t_2)] \\ \quad\quad \text{if } x_1 \mapsto x_2 \text{ and } t_1 \mapsto t_2. \end{cases}$$

The time product of a domain S creates copies of $\sigma(S)$ for each of the time steps in the time domain T. It creates arcs forward in time from each node of $\sigma(S)$ to itself and each arc gets transformed to one with the same source, but the target is the node in the next time step. This reflects the fact that any communication in space must take at least a unit of time.

A domain of the form $S * T$ will be called a *spacetime domain.*

Domain Operations

The following morphisms are used for projecting and injecting between domains of different dimensions. Let $D(i)$ be an interval domain for each i in $0 \,..\, n$.

For each k in I, the *projection* along the kth coordinate axis, the *selection* of the kth component, and the *injection* into the kth component are morphisms

$$\begin{aligned}
&\text{proj}(k) : D(0) \times \cdots \times D(n) \rightarrow \\
&\quad D(0) \times \cdots \times D(k-1) \times D(k+1) \times \cdots \times D(n) \\
&(d_0, \ldots, d_n) \mapsto (d_0, \ldots, d_{k-1}, d_{k+1}, \ldots, d_n), \\
&\text{sel}(k) : D(0) \times \cdots \times D(n) \rightarrow D(k) \\
&(d_0, \ldots, d_n) \mapsto (d_k), \\
&\text{inj}(k) : D(k) \rightarrow D(0) \times \cdots \times D(n) \\
&(d_k) \mapsto (\perp, \ldots, \perp, d_k, \perp, \ldots, \perp),
\end{aligned}$$

where $\perp$ denotes an undefined element which is less defined than any other in each index domain, and d_k is in the kth component of the tuple.

4.3 Data Fields and Data Field Morphisms

Data fields are defined to be functions from index domains to values.

Definition Let D be an index domain and V a domain of values. A *data field* over D, written $a : D \to V$, is a function $a : \sigma(D) \to V$ which assigns values to the elements of the index domain.

Example A data field $a : D \to V$ can be expressed as a function over the index domain. For example, let $D = 0 \,..\, 20$, then

$$\lambda x : D.x + 2$$

denotes a data field whose value at each index x is $x + 2$.

Index domain morphisms induce morphisms of data fields.

Definition Let $f_1 : D_1 \to V$ and $f_2 : D_2 \to V$ be data fields. A *data field morphism* g from f_1 to f_2, written $g : f_1 \to f_2$, is an index domain morphism $g : D_1 \to D_2$.

Since index domain morphisms induce data field morphisms, we shall use the term *morphism* to mean either, letting the context determine which is meant.

4.4 Dependency

A Crystal program, consisting of a set of mutually recursive definitions, determines *causal dependency* between the indices of the domains of each of the data fields.

Consider the Crystal program consisting of one definition:

$$f = \lambda x : D.h(x + 1, x + 2) + g(x - 1).$$

An *instance* of this definition is the equation obtained by applying an argument to both sides of the definition and normalizing:

$$f(2) = h(3, 4) + g(1).$$

If we consider $f(2)$ as indicating the node 2 in the domain of f, then, the node $f(2)$ is *causally dependent* on $h(3, 4)$ and $g(1)$.

By taking all possible instances of a program, we can determine the dependency of all the nodes of the data fields defined by the program. For a class of problems that is amenable to compile-time optimization, their dependency can be analyzed symbolically from the programs. Otherwise, the dependency can only be obtained during run-time [7].

4.5 Morphisms and Optimizations

Parallel-program optimization consists of reshaping the data field so that it will be most efficient for a given space of processors. Simple reshaping can be achieved by using *isomorphisms* between domains. But, in general, when there is no isomorphism, we define a pair of morphisms, called *refinement morphism*, that are almost inverses of each other,

For any morphism $g : D \to E$, a *left inverse*, if it exists, is a morphism $h : E \to D$ such that $h \circ g = 1_D$. Furthermore, if g is a left inverse of h ($g \circ h = 1_E$), then h is called an *inverse* of g and is denoted g^{-1}. Any morphism that has an inverse is called an *isomorphism.*

Definition A *refinement morphism* from D to E, denoted $(g, h) : D \to E$, consists of a domain injection $g : D \to E$, and a domain surjection $h : E \to D$ such that $h \circ g = 1_D$ and $g \circ h \sqsubseteq 1_E$. When $h = g^{-1}$, it will be called a *reshape morphism*, and will be denoted using just g.

For any injection $g : D \to E$, there are many morphisms $h : E \to D$, such that $h \circ g = 1_D$. The *conjugate* of g, denoted g^*, is the minimal such morphism under the pointwise ordering of morphisms. The conjugate maps all the nodes that are images of nodes back to their pre-image, and is undefined on all the other nodes. Clearly, for all injections g, (g, g^*) is a refinement morphism.

Example Let U_1 and U_2 be interval domains, then (g, h) is a refinement morphism from U_1 to $U_1 \times U_2$ if g maps an element u to $(u, \perp)$ and h maps (u, v) to u.

When a morphism maps some element to $\perp$, the undefined element of a domain, the implementation is free to choose some value for it to make the whole refinement optimal in some way.

Definition Let $f_1 : D_1 \to V$ and $f_2 : D_2 \to V$ be data fields. A *data field refinement* from f_1 to f_2, denoted $(g, h) : f_1 \to f_2$, is a domain refinement (g, h) from D_1 to D_2 such that $f_1 = f_2 \circ g$ and $f_2 = f_1 \circ h$.

The task of coming up with suitable morphisms is at the heart of compiler optimizations. The following is a list of commonly used reshape and refinement morphisms.

Affine Morphism

One common class of domains consists of a cartesian product of a number of interval domains with the Manhattan communication metric. Any affine injection from one such domain to another is a reshape morphism.

Example Let $D = 0 \,..\, n-1$, $E = 0 \,..\, 2n-1$, and $T = 0 \,..\, 2n-1$ be domains. Then,

$$\begin{aligned} g &: [D^2 \to E * T] \\ g &= \lambda(i, j) : D^2.(i, i + j) \end{aligned}$$

is an affine morphism.

Figure 3 shows the effect of reshaping. The space-time realization of domain D requires a time product $D * T$ while the reshaped domain $E * T$ is a space-time realization already. Hence only a linear number of processors instead of a quadratic number of processors are needed after reshaping.

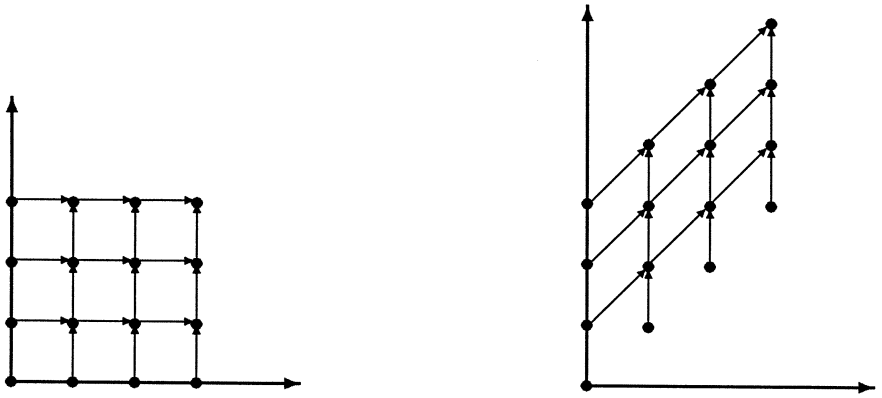

Figure 3: Affine morphism

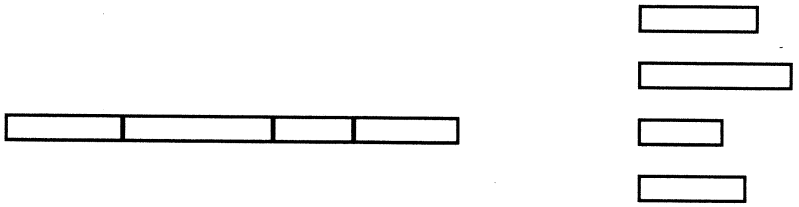

Figure 4: Partition morphism

Partition

A *uniform partition* of a domain D is a domain isomorphism $g : D \to D_1 \times D_2$. The idea is that the domain D_2 is spread over domain D_1. In general, a *partition* is an isomorphism $g : D \to \sum i : I.D(i)$, where I is another index domain, and $D(i)$ is an index domain for each i in I. Figure 4 illustrates a partition where a domain is a disjoint union of some smaller domains.

Contraction

Another class of domain morphisms comes from the idea of collapsing a domain into a smaller domain so that the data fields are folded (spliced and translated, etc.) onto the smaller domain in layers. By folding the data field in a clever way, a program requiring distant communication can be transformed into one with local communication.

For example, consider a definition for a data field that involves distant communication where the communication is symmetric with respect to some hyperplane in the index domain of the data field. The communication can be made local by defining two related data fields on the same side of the hyperplane where one has the value of the original data, except reflected along the hyperplane. Then, the two new data fields together are equivalent to the original.

Definition Let $a : D \to V$ be a data field and E a domain. A domain injection $g : D \to E + E$ with inverse $g^{-1} = [l_1, l_2] : E + E \to D$, is called a *contraction*, and the data fields $d_1, d_2 : E \to V$

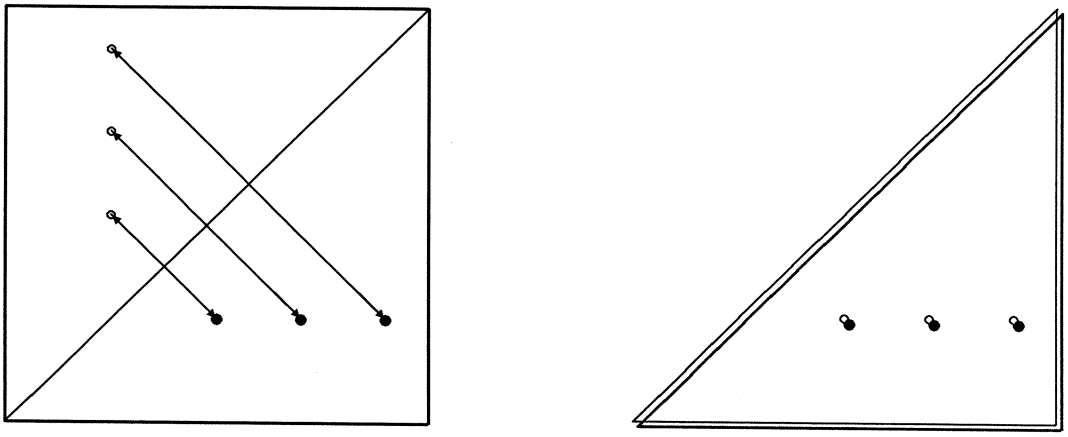

Figure 5: Contraction morphism

making

$$\begin{array}{ccc} D & \xrightarrow{a} & V \\ {\scriptstyle g}\downarrow & \nearrow_{[d_1,d_2]} & \\ E+E & & \end{array}$$

commute, are called the *layers* of the contraction.

In general, the codomain of a contraction may be the coproduct of many copies of the E's. In Figure 5, a contraction resulting in two layers is shown.

Spacetime Realization

Domains embody the logical communication costs, but before a real computation can be carried out, they need to be embedded in a spacetime domain.

Definition Let D be any index domain. A *spacetime realization* of D is an index domain injection $\langle s,t\rangle : D \to S * T$ where T is a time domain and S is a domain.

Since T is well-ordered, for any domain S, $S * T$ is well-founded. Also, since domain injections preserve paths, causal dependency in D is preserved in $S * T$.

If the domain S has suitable geometry, it makes sense to talk of the minimum area or diameter of S that realizes D, and the communication metric induced by $\langle s,t\rangle$.

Using these notions, we can characterize different objectives in the optimization of programs. For example:

1. Minimize time. Pick $\langle s,t\rangle$ to minimize the length of t.

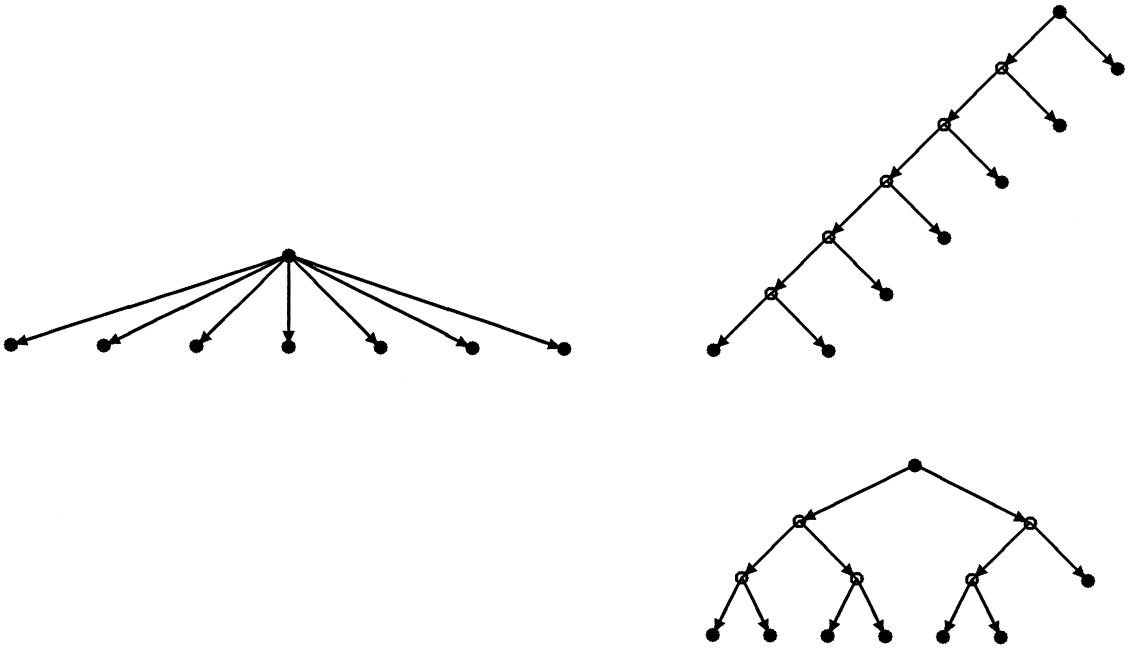

Figure 6: Fan reduction refinement

2. Maximize efficiency. Pick $\langle s, t \rangle$ to maximize the ratio of the image $\langle s, t \rangle(D)$ inside the minimum bounding box in $S * T$.

Fan-in/Fan-out Reductions

The following refinement morphisms model fan-in and fan-out reductions.

Definition For any of the domains N and S, let $D = N + r + S$ be a domain with arcs from each node of S to r, and let $E = N + \text{tree}_B(S, r)$. A *fan-in refinement* of D is a domain refinement (g, h) from D to E, where g injects N to N, and r and S to the corresponding root and leaves of $\text{tree}_B(S, r)$ and h maps r and the nodes of S in $\text{tree}_B(S, r)$ to r and S in D, N to N, and is undefined for the inner nodes of $\text{tree}_B(S, r)$.

A *fan-out refinement* is defined analogously for E containing $\text{tree}_B(S, r)$ with analogous domain refinement.

Figure 6 shows how a large fan-out can be smoothed by a left-associative tree or a balanced tree.

Currying

Currying corresponds to the abstraction of data fields so that a whole data field can be the value at an index.

For index domain D and a set of values V, let $[D \to V]$ to be the set of all data fields over D taking values in V. Using the fact that for index domains D and E and a set of values V, the function

$$\phi : [D \times E \to V] \cong [D \to [E \to V]]$$

defined by

$$\phi(f) = \lambda x.\lambda y.f(x,y) \quad \phi^{-1}(g) = \lambda(x,y).g(x)(y)$$

is an isomorphism, *currying* is the operation of converting a function f to $\phi(f)$. Interchange of loops in sequential program optimization is an example of currying.

5 Compile-time Optimizations

In Crystal, the problems of matrix multiplication and LU decomposition are specified as follows:

Program 1 (Matrix Multiplication)

$$\begin{aligned} C &= \lambda(i,j) : D. \setminus_L + [A(i,k) \times B(k,j) | k \in N], \\ N &= 1 \,..\, n, \\ D &= N \times N \end{aligned}$$

Program 2 (LU-Decomposition)

$$\begin{aligned} \textit{lu-decomposition}(A0, n) &= [L, U] \\ &\quad \textsf{where } \{ \\ a(i,j,k) : D_a &= \left\{ \begin{array}{l} k = 0 \to A0[i-1, j-1], \\ 0 < k \to a(i,j,k-1) - L(i,k) * U(k,j) \end{array} \right\}, \\ L(i,k) : D_l &= \left\{ \begin{array}{l} i < k \to 0, \\ i = k \to 1, \\ k < i \to a(i,k,k-1)/U(k,k) \end{array} \right\}, \\ U(k,j) : D_u &= \left\{ \begin{array}{l} j < k \to 0, \\ k \le j \to a(k,j,k-1) \end{array} \right\} \\ &\quad ! \textit{ define a 3-dimensional domain} \\ D &= \{1 \,..\, n\} \times \{1 \,..\, n\} \times \{1 \,..\, n\}, \\ &\quad ! \textit{ projection of } D \textit{ along the 1'st dimension} \\ D_l &= \text{proj}(1)(D), \\ &\quad ! \textit{ projection of } D \textit{ along the 0'th dimension, then transpose the domain} \\ D_u &= \text{trans}(\text{proj}(0)(D)), \\ &\quad ! \textit{ coproduct of two domains} \end{aligned}$$

$$D_a = D + (\text{proj}(2)(D) \times \{0\}),$$
$$\}$$

From this level of description, target C code plus communication commands are compiled. We illustrate in the following some of the optimizations performed by the compiler. Program Matrix Multiplication is transformed into a target Crystal program, which in turn yields the blocked partition program described in Section 6.

5.1 Fan-In reduction

In Program 1, hot spots occurs at each index $(i,j) \in D$ since n terms are summed together by the reduction operator. A fan-in reduction is performed for each occurrence of a reduction operator over a data field.

A fan-in or fan-out reduction can be achieved by using left associative tree (implemented as a linear array), a balanced tree, or a butterfly. Since our target machine is a hypercube multiprocessor on which all three structures can be emulated, a heuristic is needed to choose a particular reduction method. The number of processors and time to reduce a fan-in of n by by using tree_L is a single processor and n steps. It will take n processors and $\log n$ steps to do the same by using tree_b or the butterfly. Since the data field D is two dimensional, using either tree_B or the butterfly to reduce fan-in requires n^3 processors while using tree_L requires only n^2 processors. Given the communication metric of the target hypercube machine, the performance model described in Section 6.1 determines which method should be chosen. Generally speaking, a program with a 3 dimensional data field performs better only if the the communication latency is reasonably small and data size is very large. Note also the simple fact that it is much more convenient, and therefore desirable, that we choose the dimension of a data field so that it divides the dimension of the hypercube. For our target machine iPSC/1, the memory on each processor limits how large the problem size can be and thus the method tree_B is chosen.

Thus, we perform a fan-in reduction $\textsf{fan-in-red}(\textsf{def}(\ulcorner C \urcorner), \ulcorner Z_L \urcorner, \ulcorner \widehat{C} \urcorner)$ and the new program becomes:

$$C = \lambda(i,j) : D.\widehat{C}(i,j)(n)$$
$$\textsf{where} \left\{ \begin{array}{l} \widehat{C} = Z_L[\widehat{C}, D, H, N_0, r, +], \\ H = \lambda(i,j).\lambda k.A(i,k) \times B(k,j), \\ N_0 = 0 \,..\, n \end{array} \right\}$$

Expanding the definitions of the Z_L, we obtain

$$\widehat{C} = \lambda(i,j) : D.\lambda k : \text{tree}_L(N_0, r).\left\{ \begin{array}{l} \text{leaf}(k) \rightarrow H(i,j)(k), \\ \neg\text{leaf}(k) \rightarrow \widehat{C}(i,j)(\text{left}(k)) + \widehat{C}(i,j)(\text{right}(k)) \end{array} \right\}$$

Expanding the definitions of particular implementations of leaf, left and right of $\text{tree}_L(N_0, r)$ (where

a left-associative tree is collapsed into an array N_0), and the definition of H, we obtain

$$\widehat{C} \;=\; \lambda(i,j):D.\lambda k:N_0.\left\{\begin{array}{l} k=0 \rightarrow 0, \\ \neg(k=0) \rightarrow \widehat{C}(i,j)(k-1)+A(i,k)\times B(k,j) \end{array}\right\}$$

5.2 Partition Morphism

The next step is to adjust the granularity of a parallel computation so as to balance the communication and computation. A data field can be partitioned into a collection of sub-fields where each sub-field has a sequential space-time realization. A *simple* partition domain morphism divide a domain into subfields each of size b or less.

$$\begin{array}{rcl} h_b & = & \lambda i:N.(i \operatorname{div} b, i \bmod b):U_b \times V_b, \\ h_b^{-1} & = & \lambda(u,v):U_b \times V_b.u \times b+v:N, \\ U_b & = & 1\,..\,n \operatorname{div} b, \\ V_b & = & 0\,..\,b-1 \end{array}$$

A *compound* partition morphism is defined as the *product* of two simple partition morphisms, where the product of two functions f and g is defined as $f \times g = \lambda(i,j).(f(i),g(j))$.

The morphism we are going to apply to the matrix multiplication example is the pair of functions:

$$\begin{array}{rcl} g & = & h_{b0} \times h_{b1} \\ & = & \lambda(i,j):D.((i \operatorname{div} b0, i \bmod b0),(j \operatorname{div} b1, j \bmod b1)):(U_{b0} \times V_{b0}) \times (U_{b1} \times V_{b1}) \\ g^{-1} & = & h_{b0}^{-1} \times h_{b1}^{-1} \\ & = & \lambda((u0,v0),(u1,v1)):(U_{b0} \times V_{b0}) \times (U_{b1} \times V_{b1}).(u0 \times b0+v0, u1 \times b1+v1):D \end{array}$$

By using the meta-language operator $\mathsf{reshape}(\mathsf{def}(\ulcorner\widehat{C}\urcorner), \ulcorner g\urcorner, \ulcorner\tilde{c}\urcorner)]$, we obtain

$$\begin{array}{rcl} \tilde{c} & = & \lambda((u0,v0),(u1,v1)):(U_{b0} \times V_{b0}) \times (U_{b1} \times V_{b1}).\lambda k:N_0. \\ & & \left\{\begin{array}{l} k=0 \rightarrow 0, \\ \neg(k=0) \rightarrow \tilde{c}((u0,v0),(u1,v1))(k-1) \\ \qquad\quad +A((u0 \times b0+v0),k) \times B(k,(u1 \times b1+u1)) \end{array}\right\} \end{array}$$

5.3 Fan-Out Reduction

Note that in the definition of $\hat{c}$, $u1$ is a bound variable but it does not appear in the arguments of A on the right-hand side. Such a syntactic cue indicates that the same value $A((u0 \times b0+v0),k)$ is used by $\hat{c}(u0,u1)$ for all $u1$ in the domain U_{b1}. Thus a fan-out reduction is needed. Similarly, $u0$ is a bound variable but it does not appear in the arguments of B. Essentially, the fan-out reduction takes care of the distribution of the matrix elements of A and B across partitions.

Rather than do the fan-out reduction for the program $\hat{c}$, which get quite complex, we illustrate the method on $\widehat{C}$:

$$\widehat{C} \;=\; \lambda(i,j):D.\lambda k:N_0.\left\{\begin{array}{l} k=0 \rightarrow 0, \\ \neg(k=0) \rightarrow \widehat{C}(i,j)(k-1)+A(i,k)\times B(k,j) \end{array}\right\}$$

We will do the butterfly fan-out reduction explicitly for A, noting that a corresponding reduction can be done for B.

Since only one variable j does not occur in $A(i,k)$, we begin by separating D into its component parts, giving them distinct names.

Let $M_C = N$, $N_C = N$, and $D = M_C \times N_C$. We curry $\widehat{C}$:

$$\tilde{C} \;=\; \lambda i:M_C\lambda j:N_C.\lambda k:N_0.\left\{\begin{array}{l} k=0 \rightarrow 0, \\ \neg(k=0) \rightarrow \tilde{C}(i)(j)(k-1)+A(i,k)\times B(k,j) \end{array}\right\}.$$

In order to apply the butterfly fan-out reduction we need to change the shape of A. Let $M_A = N_A = M_B = N_B = N$ and curry A and B:

$$\begin{aligned} \tilde{A} &= \lambda i:M_A.\lambda k:N_A.A(i,k) \\ \tilde{B} &= \lambda j:N_B.\lambda k:M_B.B(k,j) \end{aligned}$$

Using the curried forms of A and B, define

$$\begin{aligned} a &= \lambda i:M_A.\lambda j:N_C.\lambda k:N_A.\tilde{A}(i)(k) \\ b &= \lambda j:N_B.\lambda i:M_C.\lambda k:M_B.\tilde{B}(j)(k) \end{aligned}$$

so that the butterfly fan-out reduction can be applied. The results of **butterfly-red**(**def**($\ulcorner a \urcorner$), $\ulcorner Q \urcorner$, $\ulcorner \bar{a} \urcorner$) and **butterfly-red**(**def**($\ulcorner b \urcorner$), $\ulcorner Q \urcorner$, $\ulcorner \bar{b} \urcorner$) are

$$\begin{aligned} \bar{a} &= \lambda i:M_A.\lambda j:N_C.\lambda k:N_A.f(j)(\dim(N_C))(k) \text{ where } \Big\{ f = Q[N_C, f, \tilde{A}(i)] \Big\} \\ \bar{b} &= \lambda j:N_B.\lambda i:M_C.\lambda k:M_B.f(i)(\dim(M_C))(k) \text{ where } \Big\{ f = Q[M_C, f, \tilde{B}(j)] \Big\}. \end{aligned}$$

Since for all i, j, and k, $\bar{a}(i)(j)(k) = A(i,k)$ and $\bar{b}(j)(i)(k) = B(k,j)$, we have

$$\bar{C} \;=\; \lambda i:M_C.\lambda j:N_C.\lambda k:N_0.\left\{\begin{array}{l} k=0 \rightarrow 0, \\ \neg(k=0) \rightarrow \bar{C}(i)(j)(k-1)+\bar{a}(i)(j)(k)\times \bar{b}(j)(i)(k) \end{array}\right\}$$

Finally, the different index domains for $\bar{C}$, $\bar{a}$, and $\bar{b}$ need to be embedded into a single index domain minimizing the communication distance. For a 2-dimensional mesh of processors with the Manhattan communication metric and with each processor possessing a local memory of length N, minimum will be when $M_A \equiv M_C$ and $N_B \equiv N_C$.

The resulting Crystal program is then translated to parallel C code.

The second example (pipelined version of matrix multiplication) described in Section 2 requires modification of the above step and a few additional optimization steps, which can be summarized as follows:

1. Instead of butterfly fan-out reduction in the above step, use tree_L.

2. Apply partition morphisms to the definitions of a and b.

3. Apply some affine morphism to achieve better utilization of processors. Linear programming is used to determine a particular affine morphism [19].

The example of LU-decomposition appears to be more complex, but essentially the same optimization steps as the pipelined matrix multiplication are required, and they are processed in the meta-language in the same way.

6 Performance

In this section, we present a performance model, which in turn defines the metric used by the compiler to choose it's optimization strategies. The performance model consists of characterization of both the program and the target machine. Characterization of the program is obtained from the source program and the optimization under consideration. Once a model is established, it is then used to predict the performance of the program on the target machine with respect to the optimization strategy.

We describe the machine profile as follows:

Machine Profile

- p – the total number of processors.

- τ_p – the time it takes to perform a unit computation on a unit data element. A unit computation may mean an integer operation, a floating point operation, or a mixed sequence of both. A unit data element is either an integer or a floating point number. The exact value τ_p depends on the instruction cycle and the basic cycle time. In this paper, all timing is given in units of micro-seconds.

- τ_c – the time it takes to transmit a unit message to an adjacent processor.

- τ_{bc} – the time it takes to broadcast a unit message to other processors. For hypercube processors,

$$\tau_{bc} = \tau_c \log(p).$$

Data Size

One of the factors that determines the amount of communication between processors is the amount of data assigned to each processor. How data is partitioned is determined by the partition morphism applied to the source program, however, the values of the parameters are often determined by optimizing the performance. Below, is a list of parameters relating to the data size.

- N – the total number of data elements in the program. For example, in Program Matrix Multiplication, N equals the total number of matrix elements.
- n – the total number of data elements on a given processor. If the data is evenly partitioned over all of the p processors, $n = N/p$.
- n_{nc} – the total number of data elements that need to be transmitted from a given processor to its adjacent processor (neighboring communications). Note that n_{nc} is a function of n.
- n_{bc} – the total number of data elements that need to be broadcast from a given processor to other processors. Similarly, n_{bc} is also a function of n.
- k_{nc} – the total number of messages for neighboring communications.
- k_{bc} – the total number of messages for broadcasting.
- $s_{nc}(j)$ – the size of the j-th message for neighboring communication.
- $s_{bc}(j)$ – the size of the j-th message for broadcasting.

And they satisfy the following:

$$n_{nc} = \sum_{j=1}^{k_{nc}} s_{nc}(j) \quad \text{and} \quad n_{bc} = \sum_{j=1}^{k_{bc}} s_{bc}(j).$$

Performance of Individual Processors

- $t_p(i)$ – computation time. It is a function of n:

$$t_p(i) = \tau_p \times n.$$

- $t_{nc}(i)$ – time for neighboring communications. In general, it is a function of n_{nc}. On the iPSC/1, $1K$ bytes is the unit message packet size, and a single precision floating-point number is 4 bytes. Thus,

$$t_{nc}(i) = \tau_c \times \sum_{j=1}^{k_{nc}} \lceil \frac{s_{nc}(j) \times 4}{1000} \rceil.$$

- $t_{bc}(i)$ – time for broadcasting. In general, it is a function of n_{bc}. On the iPSC/1,

$$t_{bc}(i) = \tau_c \times \log(p) \times \sum_{j=1}^{k_{bc}} \lceil \frac{s_{bc}(j) \times 4}{1000} \rceil.$$

- $t_l(i)$ – starting latency time. In our model, we assume that all processors start their clocks at the same time. However, if one processor needs data from other processors in order to execute, it cannot start until such data arrive. The parameter t_l is defined to be the time interval from when the clock is started until the execution begins. The interval t_l certainly varies over processors. For a pipelined algorithm, for instance, the latency, t_l, depends on the stage of a processor in the pipe. For processor i, we denote its stage by $\kappa(i)$ and the delay at each stage by Δ. Then,

$$t_l(i) = \kappa(i) \times \Delta.$$

- $t(i)$ – total elapsed time.

$$t(i) = t_p(i) + t_{nc}(i) + t_{bc}(i) + t_l(i).$$

Global Performance Parameters

- T_s – sequential elapsed time. It is defined as the total elapsed time executing the program on one processor.
- T_e – parallel elapsed time. It is defined as

$$T_e = \max_i t(i).$$

- T_p – average computation time.

$$T_p = \frac{1}{p}\sum_i t_p(i).$$

- T_c – average communication time.

$$T_c = \frac{1}{p}\sum_i (t_{nc}(i) + t_{bc}(i)).$$

- r – speedup.

$$r = T_s/T_e.$$

6.1 Performance of Fan-in Reduction Methods

As an example of using the performance model to choose optimization strategy, we compare two different fan-in reduction methods.

Suppose the input matrices are of dimension $M \times M$. Reduction tree$_L$ results in the execution of M floating point multiplication and $M-1$ floating point additions serially. Thus,

$$\begin{aligned} N &= M^2 \\ n &= N/p = M^2/p \\ \tau_p &= 25M \text{ (two } 10\mu s \text{ floating point operations and some integer index computations)} \\ T_1 &= T_s = t_p = \tau_p\, n = 25M^3/p \end{aligned}$$

To perform the fan-in reduction by tree_B or the butterfly, the processors need to be organized as a 3 dimensional data field embedded into a hypercube. We define the message size, the total number of messages, and the total communication time resulting from such a reduction as s_{red}, k_{red}, and t_{red}, respectively.

$$
\begin{aligned}
N &= M^2 \\
n &= N/p^{2/3} = M^2 p^{-\frac{2}{3}} \\
\tau_p &= 25 \\
\tau_c &= 2000 \\
t_p &= \tau_p n = 25 M^2 p^{-\frac{2}{3}} \\
s_{red} &= n = M^2 p^{-\frac{2}{3}} \\
k_{red} &= 3 \\
t_{red} &= \tau_c \log(p) k_{red} \lceil \frac{s_{red} \times 4}{1000} \rceil \\
&= 2000 \times \log(p) \times 3 \times \lceil \frac{M^2}{250 p^{2/3}} \rceil \\
&\approx 24 \log(p) M^2 p^{-\frac{2}{3}} \\
T_2 &= t_p + t_{red} = (25 + 24 \log(p)) M^2 p^{-\frac{2}{3}}
\end{aligned}
$$

For a fixed p, the two functions T_1 and T_2 intersect at some value of M, for example M_0, below which the first method should be used and beyond which the second method should be used.

6.2 Performance of Compiler-Generated Code

Several programs have been successfully compiled using our first version of the Crystal compiler. The target machine is an Intel iPSC/1 hypercube (with 32 nodes). It has since been adapted to generate code for the NCUBE hypercube (with 128 nodes). Results from both machines indicate that the performance of the compiler-generated code is comparable to those of manually-written ones. In this section, we discuss some issues of compiler-generated programs and their performances.

The Crystal compiler generates a host program to be executed on the host processor, and a node program to be executed on each node processor. To analyze the performance of the target code, we classify statements in a node program into three disjoint groups:

- *Computation statements:* Statements associated with local computation.
- *Communication statements:* Statements involving inter-processor communication. For instance, send and receive statements, statements for preparing message buffers, statements for unpacking incoming messages, etc.
- *Initialization statements:* Statements for initializing variables, array allocations, opening communication channels, etc., that are executed only once in each invocation of the node program.

The cpu time spent on the initialization statements are negligible, and can be ignored. For each processor i, the cpu time spent on the execution of a node program is collected in two parts. The *computation time* ($t_p(i)$) is the time spent on computation statements. The *communication time* ($t_c(i) + t_{bc}(i) + t_l(i)$) is the time spent on communication statements. It also includes the idle time due to waiting for other processors' messages. The *elapsed time* ($t(i) = t_p(i) + t_c(i) + t_{bc}(i) + t_l(i)$) is the time spent on the entire execution.

6.3 Performance Analysis

The performances of three compiler-generated programs are presented in this section. We briefly describe these programs, then discuss the effects of different parameters on performances.

1. **Block-partitioned matrix multiplication program.** It computes the product of two matrices, $C = A \times B$. In its execution, each processor is assigned to compute a block of the product matrix. The node program executes in two phases: a broadcasting phase and a computing phase. Initially, every processor has a block of A and a block of B. In the broadcasting phase, through row-broadcasting and column-broadcasting among processors, every processor will end up getting its needed rows and columns of A and B. In the computation phase, every processor will compute a block of C locally.

2. **Pipelined matrix multiplication program.** Here, elements of matrices A and B are pipelined through processors. The program is organized in an iteration loop. In each iteration, every processor does three things: receives some data from its neighbors, passes some data to its neighbors, and accumulates some partial results locally. The end result is that every processor will have a piece of the product matrix C.

3. **Pipelined LU decomposition program.** It decomposes a square matrix A into its L and U factors. This program is structurally the same as the pipelined matrix multiplication program.

Compiler Generated Program vs. Hand-Written Program

Figure 7 shows the performances of three matrix multiplication programs, which use the same block partitioned algorithm. Two of them are generated by the Crystal compiler. The other is manually written. One of the compiler-generated programs uses static memory allocation, i.e., array sizes are runtime constants. The other uses dynamic memory allocation, i.e. arrays are allocated by *malloc()* at runtime. The hand-written program uses static memory allocation.

As can be seen in Figure 7, the compiler-generated programs perform almost as well as the hand-written ones. The overheads introduced by the compiler are negligible.

Pipelined Communication vs. Broadcasting

These two matrix multiplication programs employ different types of communications.

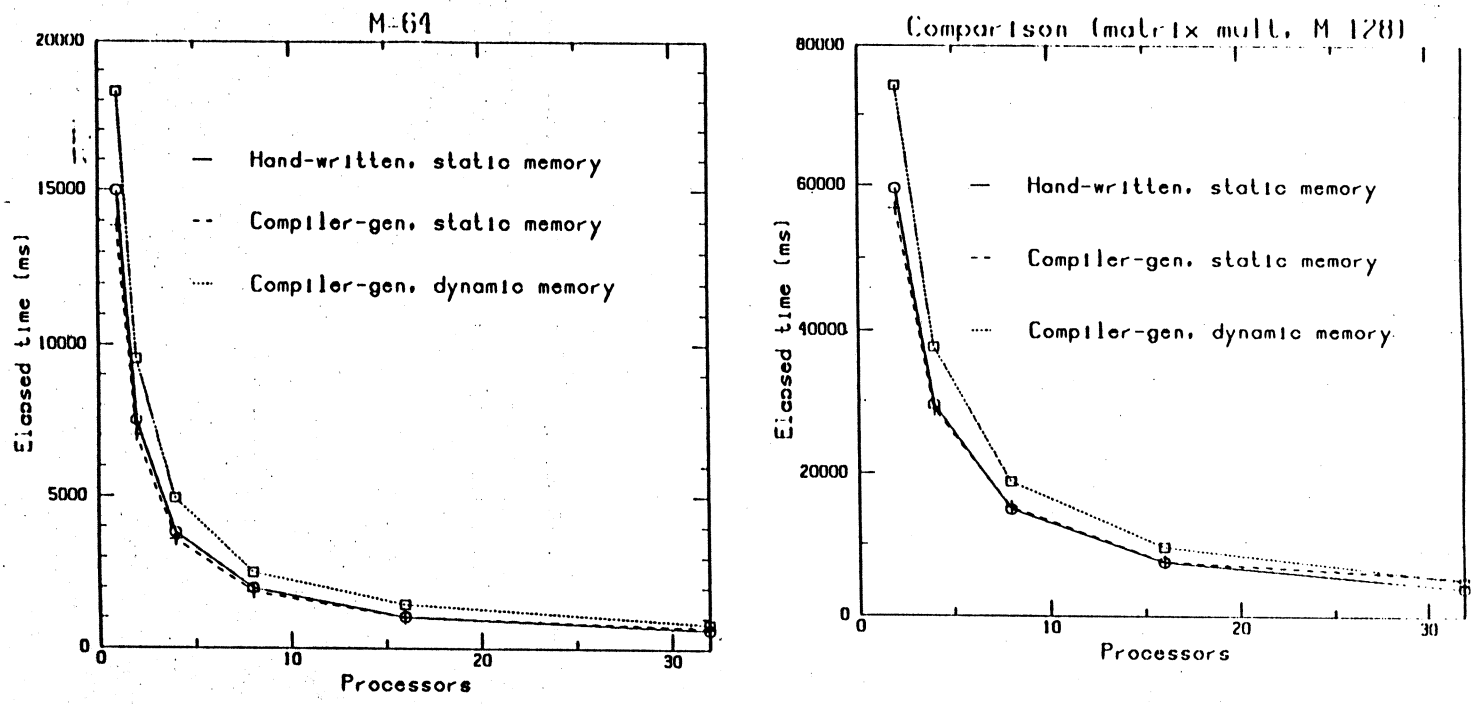

Figure 7: Performances of three block MM programs on iPSC/1. Two are generated by compiler; one is manually written.

In the pipelined version, communication is only done between adjacent processors. This locality in communication is very suitable for the iPSC and the NCUBE because on both machines the communication cost is proportional to the distance between the source and the destination processors. Another good feature of communication locality is that the communication cost does not scale up with the size of the machine. If the problem size is fixed, increasing the number of processors in a multiprocessor will reduce the sub-problem size on every processor. This reduces the amount of messages that every processor has to communicate with. Therefore, increasing the number of processors will reduce communication time for each processor.

In the block-partitioned version, broadcasting is needed among many processors (a row or a column). Broadcasting is done by dynamically creating a butterfly and exchanging data at each stage of the butterfly. The time for broadcasting is $\log(p)$, where p is the number of processors involved (see [17] for a detailed discussion about optimal broadcasting on hypercubes). When the number of processors is increased, the communication time may not decrease because even though the amount of messages on each processor will be reduced, the cost of broadcasting will increase.

In Figure 9, we see that for the pipelined program, both T_p and T_c decrease when p increases. For the block-partitioned program, T_c stays roughly at the same level, no matter what p is. Therefore, for the block-partitioned program, when p increases, a higher percentage of cpu time will be spent on communication. The speedup of parallel-elapsed-time over sequential-elapsed-time will not grow linearly with the increase of p, as shown in Figure 9.

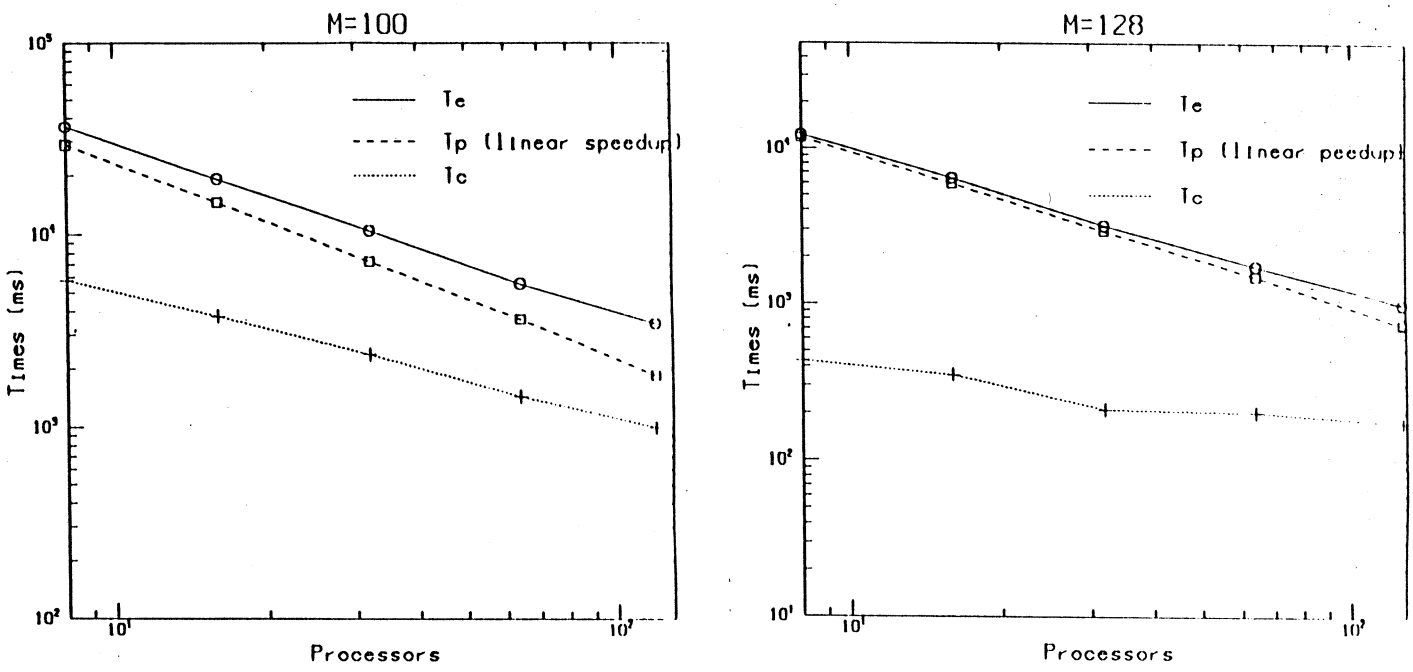

Figure 8: Performances of two different types of matrix multiplication program: pipelined and block-partitioned, on NCUBE.

Effects of Changing Granularity

When a program is organized in an iterative loop (like the pipelined matrix multiplication program or the pipelined LU decomposition program), a parameter is provided by the compiler to control the size of an iteration step between two inter-processor communications. Suppose the input matrices are of dimensions $M \times M$ and that the total computation is divided into k equal iterations. The changing of the value of k effects the overall performance of the program. When k is small, the starting-latency time is small, which is good. However, the total number of messages increases, which may introduce more communication overheads. In the following: we find the value of k that optimizes the overall performance.

$$
\begin{aligned}
N &= M^3 \\
n &= N/p = M^3/p \\
\tau_p &= 30 \\
\tau_c &= 2000 \\
s_{nc} &= n^{2/3}/k = M^2/pk \\
k_{nc} &= 4k \\
t_p(i) &= \tau_p n = 30M^3/p
\end{aligned}
$$

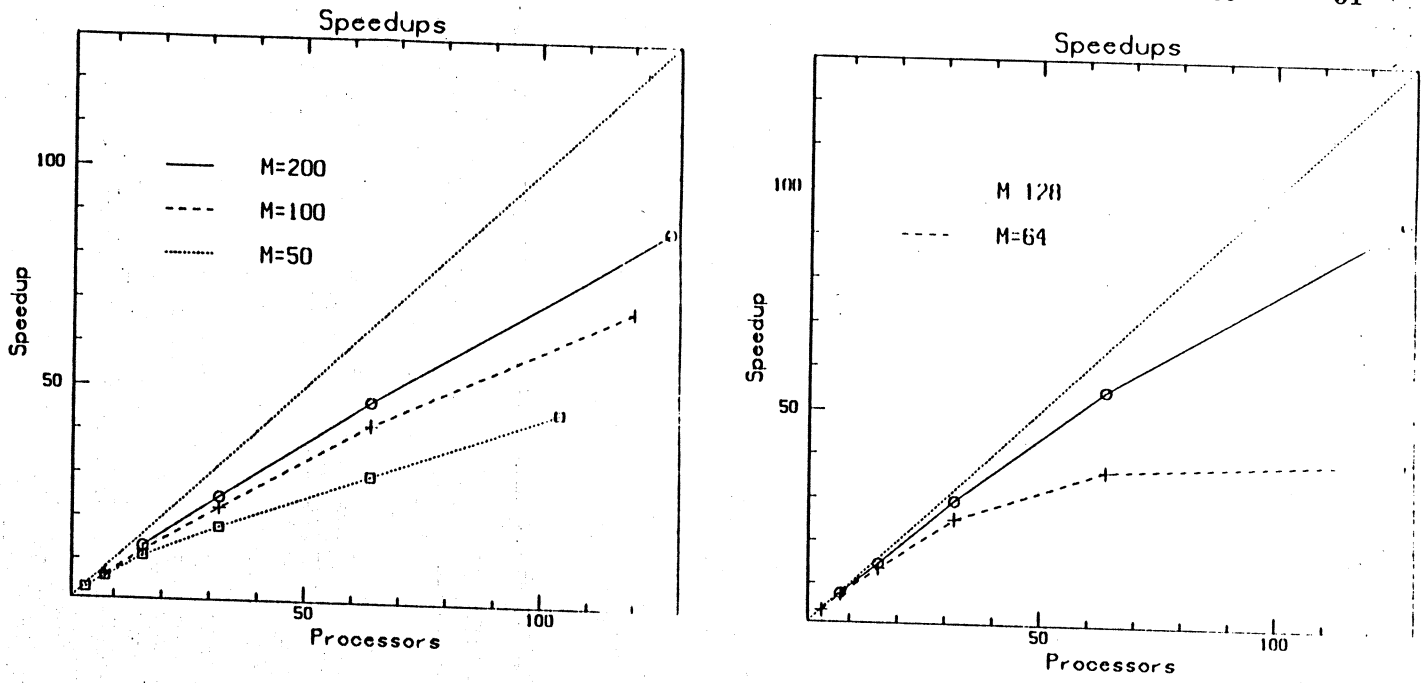

Figure 9: Speedups of the pipelined, and the block-partitioned matrix multiplication programs on NCUBE.

$$
\begin{aligned}
t_{nc}(i) &= \tau_c \sum_{j=1}^{k_{nc}} \lceil \frac{s_{nc}(j) \times 4}{1000} \rceil \\
&= 4k \times 2000 \times \lceil \frac{M^2}{250pk} \rceil \\
&\approx 8000k \qquad (\text{if } M^2/250pk < 1) \\
t_{bc}(i) &= 0 \\
t_l(i) &= \kappa(i)\tau_p/k = 30\kappa(i)M^3/pk \\
t(i) &= 30M^3/p + 8000k + 30\kappa(i)M^3/pk \\
T_e &= \max_i t(i) \\
&= 30M^3/p + 8000k + (30M^3/pk) \max_i \kappa(i) \\
&= 30M^3/p + 8000k + 60M^3/k\sqrt{p}
\end{aligned}
$$

Remarks:

1. For the pipelined version, the unit computation involves several array references and 2 floating point operations (e.g. $c[i][j][k] = a[i][j][k] * b[i][j][k] + c[i][j][k-1]$). On the Intel iPSC/1, a floating point operation costs $10\mu s$, so roughly, $\tau_p = 30\mu s$.

k	1	2	3	4	5	7	10	17	20	25	50
$e(k)$ (model)	250	110	67	44	31	16	6.6	0.2	0.0	0.9	14
$e(k)$ (experimental)	76	33	19	12	7.5	3.4	0.7	0.0	0.0	1.5	10

Table 1: Comparison of the predicted performance and the actual performance

2. On the iPSC/1, transforming a message of size 1K bytes or smaller takes $2ms$.

3. The matrix is partitioned into square blocks. All messages are made to be the same size and they consist of $n^{2/3}$ elements each.

4. In each iteration, a processor sends and receives two messages. Thus, a processor involves a total of $4k$ messages.

By symbolically differentiating T_e with respect to k, we obtain its minimum value when

$$k = \sqrt{\frac{60M^3/\sqrt{p}}{8000}} = \sqrt{\frac{3M^3}{400\sqrt{p}}}$$

and

$$T_e = 30M^3/p + 800\sqrt{\frac{3M^3}{\sqrt{p}}}.$$

For $p = 4$ and $M = 50$ we have $k \approx 20$.

In order to compare the performance predicted by the model and the actual experimental result, we define the percentage of extra time taken, for each value of k, over the optimal time:

$$e(k) = \frac{|T(k) - T(\overline{k})|}{T(\overline{k})} \times 100\%$$

where $\overline{k}$ denotes the optimal k value. Table 1 summarizes the result for $M = 50$ and $p = 4$.

iPSC vs. NCUBE

All three programs have been run on an iPSC and an NCUBE. Tables 2 and 3 list the performances on the two machines.

The performance of the target code on the largest hypercube used in our experiments is summarized in Table 4.

iPSC/1 (p=32)				
program	T_p	T_c	T_e	r
Pipelined MM	10698	3330	14042	22.7
Block MM	2103	340	2472	22.6
Pipelined LUD	11777	3455	16695	22.6

Table 2: Performances of three programs on iPSC. Matrix-size = 100 x 100.

NCUBE (p=32)				
program	T_p	T_c	T_e	r
Pipelined MM	7254	2382	10420	22.3
Block MM	1386	136	1620	27.4
Pipelined LUD	8138	2670	11860	22.0

Table 3: Performances of three programs on NCUBE. Matrix-size = 100 x 100.

NCUBE (p=128)				
program	T_p	T_c	T_e	r
Pipelined MM	14317	5075	21060	87.0
Block MM	721	171	980	94.2
Pipelined LUD	16240	5721	24220	85.8

Table 4: Performances of three programs on 128-node NCUBE. Matrix-size for pipelined MM and LUD is 200 x 200; for Block MM is 128 x 128.

7 Concluding Remark

The parallelizing compiler approach and Crystal's approach to parallel processing attack the same problem from two opposite perspectives: one attempts to increase the degree of parallelism while the other reduces it. Nevertheless, many of the techniques used in parallelizing Fortran compilers [1,9,11,20,18,3,21,22] can be understood as morphisms. For example, *strip mining*[20], a technique for memory management, transforms a single sequential loop into a double nested one so that the

outer loop of the resulting code can be executed in parallel. The partition morphism of Crystal, in contrast, transforms a one-dimensional data field, where all elements can be executed in parallel, into a two-dimensional data field. The computation along one of the dimensions can then be serialized in order to reduce communication overhead. Crystal's fanout reduction and contraction morphisms are particularly useful for distributed-memory machines where the cost of accessing data is not uniform.

Much effort has been made in the realm of functional languages [2] and data flow processing (see summary in [25]) for discovering latent parallelism in programs. Due to the freedom from side-effects, functional calls that are not interdependent can be executed in parallel. But the focus of data flow processing has been on parallel tasks with different threads of control. The distribution (copying) of data to parallel tasks therefore becomes a source of inefficiency. Remedies such as explicit specifications of data locations by annotations are used in [14]. Though Crystal is also a functional language, it tackles the data distribution problem at the most fundamental level, namely, in the model. The notions of data fields and communication metrics are essential for the compiler optimization that generates efficient code.

Because we are free from the constraints of an existing language, we can make Crystal and, most importantly, its model to be algebraic. Consequently, optimization techniques are captured as morphisms, which can be expressed as Crystal functions and carried out by the metalanguage processor in a uniform and systematic fashion. We also believe that the theoretical foundation helps us in managing the complexity of implementing Crystal by factoring out the part of the system that contains heuristics (the optimization library) and the part of system which is entirely algebraic. Currently, we are planning the programming of large, realistic applications in Crystal. We hope to demonstrate that elegance, practicality, and high performance are not competing goals but synergistic ones.

Acknowledgment We would like to thank Erik DeBenedictis for his help in our experiments on the NCUBE. Our thanks also to the referees for their many helpful comments. We would like to thank Eileen Connolly for her editorial efforts. The generous support by the Office of Naval Research under Contract No. N00014-86-K-0564 is gratefully acknowledged.

References

[1] J .R. Allen and K. Kennedy. *Supercomputers: Design and Applications*, chapter PFC: a program to convert Fortran to parallel form, pages 186–205. IEEE Computer Society Press, 1985.

[2] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[3] Utpal Banerjee, Shyh-ching Chen, and David J. Kuck. Time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computers*, C-28(9):660–670, September 1979.

[4] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[5] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[6] M. C. Chen. Very-high-level parallel programming in Crystal. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[7] M. C. Chen and M. Jacquemin. *Footprint of Dependency: Towards Dynamic Memory Management for Massively Parallel Architectures.* Technical Report 593, Department of Computer Science, Yale University, January 1988.

[8] Young-il Choo and Marina Chen. *A Theory of Parallel-Program Optimization.* Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.

[9] R.G. Cytron. Doacross: beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing (St. Charles, Ill. Aug.19-22)*, pages 836–844, IEEE Press, New York, August 1986.

[10] John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[11] J.A. Fisher, F.R. Ellis, J.C. Ruttenberg, and NIcolau A. Parallel processing: a smart compiler and a dumb machine. In *ACM-Sigplan 84 Compiler Construction Conference*, ACM, June 1984.

[12] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Springer-Verlag, 1979.

[13] Frank Harary. *Graph Theory.* Addison-Wesley, 1969.

[14] Paul Hudak. Para-functional programming: a paradigm for programming multiprocessor systems. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[15] Paul Hudak. Private communications.

[16] Kenneth E. Iverson. *A Programming Language.* John Wiley and Sons, New York, 1962.

[17] S. L. Johnsson and C.-T. Ho. *Matrix Multiplication on Boolean Cubes Using Generic Communication Primitives.* Technical Report TR-530, Yale University, 1986.

[18] David Kuck. A survey of parallel machine organization and programming. *Computing Survey*, 9(1):29–59, March 1977.

[19] J. Li, M.C. Chen, and M.F. Young. *Design of Systolic Algorithms for Large Scale Multiprocessors.* Technical Report 513, Yale University, 1986.

[20] D. B. Loveman. Program improvement by source-to-source transformation. *JACM*, 24(1):121–145, January 1977.

[21] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[22] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of ACM*, 29(12):1184–1201, December 1986.

[23] Brian Cantwell Smith. Reflection and semantics in lisp. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, ACM, Salt Lake City, Utah, January 1984.

[24] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1979.

[25] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Computing Surveys*, 14(1):93–143, March 1982.