Yale University
Department of Computer Science

A Theory of Parallel-Program Optimization

Young-il Choo and Marina C. Chen

YALEU/DCS/TR-608
July, 1988

A Theory of Parallel-Program Optimization

Young-il Choo Marina Chen

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

July 18, 1988

Abstract

In this paper we present a theory of parallel program optimization. We begin with a language; construct a model of parallel computation; define an equational theory of the language; and present a metalanguage for formalizing the program transformations. To illustrate these ideas, we show detailed examples of the derivation of optimized programs using the formal transformations.

The central notion in our model is that of a *data field*, a distribution of data values over a space of processors known as an *index domain*. A parallel program, presented as a system of mutually recursive definitions, denotes a set of interdependent data fields called a *computation field*. The reshaping of data fields is represented by *data field morphisms*.

The heart of our theory is the connection between the semantic notion of the data field morphism and the syntactic operation of the program transformation. Given a program which denotes a data field the creative task is to find a data field morphism that produces a more efficient data field. Using the definition of the data field morphism, a new program can be derived within the equational theory of Crystal that has the new data field as its denotation.

Contents

1 Introduction

Decades of experience has impressed upon us the complexity of programming computers. Making the programs correct is hard enough, making them run efficiently requires more work. With the arrival of large-scale multi-processor computers, we now face the challenge of parallel programming where the allocation of processors and the cost of inter-processor communication enter into the question of efficiency. Unless this complexity can be managed, the task of distributing the computation over a space of processors will simply replace the von Neumann architecture in sequential computers as the new bottleneck to cost-effective computation.

For sequential programming languages, program transformation has been useful for making the programming task easier. The idea being to begin with a clear and concise program and then transform it in well defined steps to produce an equivalent version that is more efficient. The most common goals of program transformation are the optimization of program control structures (using *folding* and *unfolding*, for example [6,22,23]) and the efficient implementation of composite data structures (like *set* and *map* types in SETL).

Experience with the design of parallel programs indicates that program transformation is an indispensable tool in deriving programs that coordinate parallel tasks with intricate space and time dependencies [4]. In this paper we present a theory of parallel program optimization. We begin with a language; construct a model of parallel computation; define an equational theory of the language; and present a metalanguage for formalizing the program transformations. To illustrate these ideas, we show detailed examples of the derivation of optimized programs using the formal transformations.

The programming language we use is Crystal—a functional language with composite data structures. We freely use the lambda-notation for representing functions, which turns out to be invaluable, if not essential, in formalizing various program transformations. Our choice of a functional language lies in our view that parallelism is an interpretation of the language on specific models of computation, and that it should be implicit in the program with only the data dependencies being specified, leaving the implementation to map the computation to processors in space and time. We restrict ourselves to deterministic programs in this paper.

In our presentation of the model of Crystal, we assume familiarity with the standard denotational semantics of functional languages where the meaning of a recursively defined function is given as the minimal fixed-point of the defining functional [28]. What is new is the interpretation of the domain of certain functions as the spacetime domain of processors and the additional structure on the domain for representing the data dependencies and the communication cost between the processors.

The basic model is built upon *index domains* which represent the location of processors. A function call with a certain index value is interpreted as communication with a processor at the location labeled by the value or index. By defining a suitable topology and communication metric on an index domain, we can model different types of parallel machines, from shared memory to distributed memory machines where the processors are connected in a network such as a mesh or hypercube etc.

Composite data structures are represented as functions over index domains and are called *data fields* to emphasize their role as distributed data structures. The shape of a

data field is the topology and the communication metric of its index domain. Mappings from one index domain to another which preserves the topology may be used to change the shape of data fields. This is captured in what we call *data field morphisms.*

In the standard program transformation method for sequential programs, the transformations are only done on the expressions of the programs and are denotation preserving. In our approach, we consider program definitions as equations in an equational theory of the language and do transformations on equations, not just expressions. This generates a larger class of transformations, including ones which produce new programs with different denotation, albeit, related to the former in a well defined way. In order to formalize these transformations, we define a metalanguage for constructing and modifying Crystal programs.

The heart of our theory of program optimization is the connection between the semantic notion of the data field morphism and the syntactic operation of the program transformation. Given a program which denotes a data field the creative task is to find a data field morphism that produces a more efficient data field. Using the definition of the data field morphism, a new program can be derived within the equational theory of Crystal that has the new data field as its denotation.

To be more specific, an initial problem specification is often a program with a straightforward structure of distributed data which denotes a data field a. An efficient solution of the problem often needs data to be distributed in a different, more sophisticated way, described by a new program denoting the data field $\hat{a}$. The transformation from the initial program to one that is more efficient changes the denotation of the program (from data field a to data field $\hat{a}$). The relationship between the old data field a and the new data field $\hat{a}$ is represented by a mapping, called an *index domain morphism*, between the old index domain D of a and the new index domain $\hat{D}$ of $\hat{a}$.

This idea of data field morphisms can be extended to map abstract computation onto specific computer architectures. A *spacetime realization* of a parallel computation gives us a metric for measuring the communication cost, the number of processors required, and the time steps required.

To make these ideas more concrete, we present three examples of program optimization based on this framework. The derivations are given as a sequence of steps, defined in the metalanguage, which produces the new program when given the initial program and the definition of the index domain morphism.

Approaching parallel program optimization by program transformations allows the program to be written initially in a logically clear form without worrying about efficiency. The more efficient version may then be produced by a transformation system according to the necessary transformations specified in the metalanguage. The set of parameters that come into play in designing parallel programs has become so large that we believe program transformation will become an essential part of parallel program development. We believe, in particular, that the data field morphism which describes the assignment of logical data structure to physical processors, will be at the heart of many parallel program design efforts.

The task of parallel program optimization consists then in finding suitable index domain morphisms which will increase the efficiency of the communication structure of the original program. The actual derivation of the new program turns out to be merely a sequence of mechanical operations on the program.

The rest of this paper is organized as follows. The Crystal language is briefly introduced in Section 2. Section 3 introduces the model of parallel programs. In Section 4, an equational theory of the Crystal language is presented along with the metalanguage for specifying the program transformations. Using the framework introduced, we discuss the overall strategy for optimization in Section 5. Section 6 contains examples of data field morphisms and the transformation steps defined using the operators of the metalanguage. A few concluding remarks are given in Section 7.

2 The Crystal Language

The objects of Crystal include index domains, domain morphisms, data fields, and computation fields. These will be formally defined in Section 3. The syntax of Crystal is basically the the notation for objects of the semantics with 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.

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 $a : D \rightarrow V$ 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 ([12]) 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 ([12]) is defined similarly

$$\backslash\backslash f\,[a_0, \ldots, a_n] \;=\; [a_0, f(a_0, a_1), \ldots, f(\cdots(f(f(a_0, a_1), a_2))\cdots)]$$

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 A Model of Parallel Computation

In this section we present a model of parallel computation that forms the foundation of an equational theory of program optimizations.

The central notion in our model is that of a *data field*, a distribution of data values over a space of processors known as an *index domain*. A parallel program, presented as a system of mutually recursive definitions, denotes a set of interdependent data fields called a *computation field*. The reshaping of data fields is represented by *data field morphisms*.

The algebra of morphisms of the data fields provide us with an equational theory in which new types of transformations can be defined to improve the overall efficiency of parallel programs.

Index domains are given a topology by defining a communication metric which represents communication cost between the 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.

3.1 Index Domains and Data Fields

In this section we present the objects which model parallel-programs and program transformation. The basic objects are the *data fields*. Informally, a data field represents the distribution and data dependence of data values over some domain of processing nodes. An index domain represents the communication paths and data dependency between the nodes of the domain.

Communication Metric

Before defining index domains, we need the notion of a communication metric which represents the cost of communication between nodes in a space of processors.

Let R be the set of non-negative real numbers with positive infinity.

Definition A *communication metric* on a set S is a function $\gamma : S \times S \to R$ satisfying the following:

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

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

Note that a communication metric is strictly weaker than the usual geometric notion of a distance metric [7], which also is symmetric (for all x and y, $\gamma(x, y) = \gamma(y, x)$) and satisfies the converse of the first condition ($\gamma(x, y) = 0$ implies $x = y$). This weaker notion is motivated from the observation that communication costs in opposite directions need not be equal, and that several logical processes may be mapped onto a single physical one resulting in the communication cost being zero.

Index Domains and Index Domain Morphisms

Definition An *arc* over a set S is an ordered pair of elements from S, denoted $x \mapsto y$, where x is called the *source* and y the *target* of the arc. 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.

We define index domains to be directed graphs with extra structure.

Definition An *index domain*, D, is a structure

$$\langle \sigma(D), \rho(D), \gamma(D) \rangle,$$

where $\sigma(D)$ is a non-empty set, $\rho(D)$ is a set of arcs over $\sigma(D)$, and $\gamma(D)$ is a communication metric over $\sigma(D)$.

Just as for graphs, an index domain has *finite degree* if each node has a finite number of incoming and outgoing arcs. An index domain is *well-founded* if no path can be extended at its source infinitely often.

Next, we define the notion of mappings, or morphism, between index domains. The crucial feature of such a mapping is that it must preserve paths, since paths represent possible communication paths.

Definition An *index domain injection*, written $g : G \to H$, is an injection $g : \sigma(G) \to \sigma(H)$ that preserves paths: if $x \overset{*}{\mapsto} y$ in G, then $g(x) \overset{*}{\mapsto} g(y)$ in H. The *identity* on an index domain G, denoted 1_G, is an injection that is the identity on the nodes.

An *index domain surjection* is a (possibly empty) sequence of elementary surjections, where an *elementary surjection* is a mapping from one index domain to another that collapses one arc with its source and target into one node while preserving paths (In [11] these are called elementary homomorphisms).

A *index domain morphism* is a combination of index domain injections and surjections.

We allow index domain morphisms to be partial functions, denoting the undefined value at a node by $\perp$. This is useful for when we want the system to choose some value based on optimization. To do this rigorously, we can define the nodes of an index domain to be a flat lattice of nodes with the undefined node ($\perp$) less than any other node in the ordering of the lattice ($\sqsubseteq$). For sake of simplicity we do not do this.

Definition For any index domain morphism $g : G \to H$, a *left inverse*, if it exists, is a morphism $h : H \to G$ such that $h \circ g = 1_G$. If g is also a left inverse of h (i.e., $g \circ h = 1_H$), then h is called an *inverse* of g and is denoted g^{-1}. Any morphism that has an inverse is called an *isomorphism.*

Note that though an index domain is a structure with three components, in Crystal programs an index domain is normally defined by its set of nodes. The arcs are given by the data dependencies implicit in the program and the communication metric will usually be induced by another index domain. These will be defined below. Also, the nodes are more than a set, they are actually elements of an algebra, with operations and predicates defined over them. So, for example, when the nodes consist of the integers, we assume that all the arithmetic operations are defined for them.

Since an index domain morphism preserves paths between two index domains, we can use it to induce a communication metric in the source index domain from the communication metric of the target index domain.

Definition Let $g : D \to E$ be a domain morphism and $\gamma(E)$ be a communication metric over E. The communication metric on D *induced* by g is

$$\gamma_g(D) = \lambda(x, y).\gamma(E)(g(x), g(y)).$$

Here, the index domain D can be thought of as a sub-domain of E.

Sometimes it is more intuitive to give the communication cost on each arc and then let the communication metric be induced from them. Call a function $\nu = \nu(G) : \rho(G) \rightarrow R$ that assigns to each arc a communication cost, a *local metric* on G. Given a local metric, which gives the communication cost for each arc, it is straight forward to extend it inductively to a path so that for any path p, $\nu(p)$ is the sum of the cost of each arc in the path. Using this extension we define the following function which will be shown to be communication metric.

Definition Let ν be a local metric on G. Define the function $\nu^* : \sigma(G) \times \sigma(G) \rightarrow R$ as follows:

1. $\nu^*(x, y) = 0$, if $x = y$.
2. $\nu^*(x, y) = \nu(x \mapsto y)$, for all arcs $x \mapsto y$.
3. If $x \overset{*}{\mapsto} y$, then $\nu^*(x, y) = \min\{\, \nu(p) \mid \ p$ is a path from x to y. $\}$.
4. $\nu^*(x, y) = \infty$, if there is no path from x to y.

The *default* communication metric on G is one induced by the local metric that assigns to each arc one unit of communication cost.

We next present several examples of index domains.

Definition An *interval index domain*, denoted $l \,..\, u$, is an index domain whose nodes are the integers between l and u, inclusively. 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.

Definition A *time domain* is a linear, well-founded graph with the default communication metric. In particular T_n will denote a time domain of length n.

Given any well-founded index domain, there are many ways of linearizing it. 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. Since domain injections preserve paths, the linearization must be consistent with the ordering in the original D. Note that $\tau(D)$ represents an arbitray linearization of D.

Any set can be made into a domain by making it into a complete graph with unit communication cost between any two nodes. For any set S, the *complete graph domain*, denoted $\delta(S)$, is a domain with nodes from S forming a complete graph with unit communication cost between any two nodes.

Index domains can be of general shape. The following define a tree connected index domain.

Definition 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, left child, and right child of each node, respectively, if they exist, and is undefined otherwise.

Let S be a set of nodes and r a node not in S.

Definition 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. The arcs are pointed from the children nodes to their parent. 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.

A tree domain denoted $\text{tree}(r, S)$ has the same nodes but the arcs point from the parent to their children.

Index Domain Constructions

Various domain constructions will be presented. The usual cartesian product and disjoint union are defined, and a special construction called the time product is introduced. Subdomains can be defined by selecting certain elements which satisfy a boolean function.

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 by

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

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 occur only along directions parallel to the axes.

Next, we define the coproduct, or disjoint union. In the coproduct, there can be new arcs that are added between nodes of the different components. These arcs are given separately as the *glue* of the coproduct. Given two domains, a *glue* between them is a set of arcs linking nodes from one to the other domain.

Definition Let D_1 and D_2 be domains and let J be a glue between them. The *coproduct* of D_1 and D_2 with glue J is a domain $D_1 +_J D_2$, with two injections $\iota_1 : D_1 \to D_1 +_J D_2$ and $\iota_2 : D_2 \to D_1 +_J D_2$, defined by

$$\begin{aligned}
\sigma(D_1 +_J D_2) &= \{\, \iota_1(x) \mid x \in \sigma(D_1) \,\} \cup \{\, \iota_2(x) \mid x \in \sigma(D_2) \,\} \\
\rho(D_1 +_J D_2) &= \rho(D_1) \cup \rho(D_2) \cup J
\end{aligned}$$

where ι_1 and ι_2 are the encoding injections of the nodes to make them disjoint. The communication metric is induced by the new communication cost of the arcs in the glue with the separate communication cost of each component.

The injections ι_1 and ι_2 abstractly indicate into which summand an element is mapped. The most interesting use of the coproduct is when the components are the same. In this case the default inter-component communication cost is defined to be zero: $\gamma(E +_J E)(\iota_1(x), \iota_2(x)) = 0$ for all x in domain E.

Definition Let D_1 and D_2 be domains and $l_1 : D_1 \to E$ and $l_2 : D_2 \to E$ be a pair of functions. The *coproduct map* of l_1 and l_2, denoted $[l_1, l_2] : D_1 + D_2 \to E$, is a function equivalent to l_1 if the argument is from D_1 and equivalent to l_2 if the argument is from D_2. Algebraically, it satisfies the following:

$$l_1 = [l_1, l_2] \circ \iota_1 \quad \text{and} \quad l_2 = [l_1, l_2] \circ \iota_2.$$

The product and coproduct constructions may be generalized to arbitrary set of domains. Let I be a domain and D_i be a domain for each i in I.

Definition The *product* of a set of domains $\{ D_i \mid i \in I \}$ is a domain, denoted $\prod i : I.D_i$, with projections $p_k : \prod i : I.D_i \to D_k$ for each i in I.

The *coproduct* of a set of domains $\{ D_i \mid i \in I \}$ with glue J is a domain, denoted $\sum_J i : I.D_i$, with injections $\iota_k : D_k \to \sum_J i : I.D_i$ for each i in I.

The coproduct allows the construction of arbitrarily shaped domains. Nested loops can be regarded as evaluation over coproduct domains.

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 can only point forward in time.

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

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

where

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

Time product of a domain S creates copies of $\sigma(S)$ for each of the time steps in the time domain T and creates arcs forward in time from each node of $\sigma(S)$ to itself and each arc gets transformed to one with same source, but the target is in the next time step. This reflects the fact that any communication in space must take at least a unit of time. Figure 1 illustrates the difference between the cartesian product and the time product.

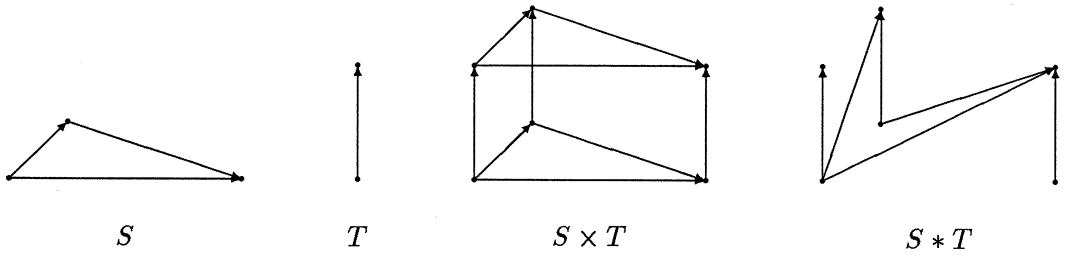


Figure 1: The cartesian and the time products of S and T.

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

Another method for obtaining a new domain is by filtering the nodes of a previously defined domain. A filter is simply a boolean function over the nodes of the domain.

Definition Let D be a domain and P be a filter over D. The *restriction* of D by P, denoted $D{\downarrow}P$, is a domain whose nodes are the nodes of D satisfying P, and whose arcs are given by the following: (1) if an arc exists between two nodes satisfying P, the the arc is in the restriction, (2) if there is a path between two nodes satisfying P but all the intermediate nodes do not satisfy P, then we add a new arc between these two nodes.

Index 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 $0 \,..\, n$.

Definition The *projection*, *selection*, *injection*, and *transpose* functions are defined as follows:

$$\begin{aligned} \text{proj}(k) &: D_0 \times \cdots \times D_n \to D_0 \times \cdots \times D_{k-1} \times D_{k+1} \times \cdots \times D_n \\ &\quad (d_0, \ldots, d_n) \mapsto (d_0, \ldots, d_{k-1}, d_{k+1}, \ldots, d_n), \end{aligned}$$

$$\begin{aligned} \text{sel}(k) &: D_0 \times \cdots \times D_n \to D_k \\ &\quad (d_0, \ldots, d_n) \mapsto (d_k), \end{aligned}$$

$$\begin{aligned} \text{inj}(k) &: D_k \to D_0 \times \cdots \times D_n \\ &\quad (d_k) \mapsto (\underbrace{\perp, \ldots, \perp}_{k}, d_k, \perp, \ldots, \perp), \end{aligned}$$

$$\begin{aligned} \text{trans}(i,j) &: D_0 \times \cdots \times D_i \times \cdots \times D_j \times \cdots \times D_n \to D_0 \times \cdots \times D_j \times \cdots \times D_i \times \cdots \times D_n \\ &\quad (d_0, \ldots, d_{i-1}, d_i, \ldots, d_{j-1}, d_j, \ldots, d_n) \mapsto (d_0, \ldots, d_{i-1}, d_j, \ldots, d_{j-1}, d_i, \ldots, d_n) \end{aligned}$$

where $\perp$ denotes an undefined element in each index domain.

Data Fields and Data Field Morphisms

Data fields model parallel computation and are functions which assigns values to indices of a domain. In the following, the domain of values will usually be the integers or the reals, but it can also recursively be a domain of data fields and other higher order objects.

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 a value to each node of the index domain. The domain of data fields over D with domain of values V is denoted $[D \to V]$. And we say that data field a is of *type* $[D \to V]$.

The domain of values, V, will typically be the integers, the reals. Higher-order data fields will have other domains of data fields as their domain of values.

To be more precise, we can define the domain of all data fields, F, using lattice theoretic domain equations. Let $\mathcal{D}$ be the set of all index domains, and let P be the union of all the primitive value domains (the integers, the reals, the boolean values).

$$\begin{aligned} F &= \bigcup_{D \in \mathcal{D}} [D \to V] \\ V &= F + P. \end{aligned}$$

(This definition actually contains more than we need since it allows for F to be reflexive. All we need is the union of all the inductively defined levels.)

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 $x + 2$. More complex data fields can be defined using a suitable language which will be presented later.

Definition Let $[D_1 \to V_1]$ and $[D_2 \to V_2]$ be domains of data fields. A *data field morphism* is a map

$$\phi : [D_1 \to V_1] \to [D_2 \to V_2].$$

This definition is very general. We introduce two types of data field morphisms that will be useful later. The first type comes from the observation that each index domain morphism defines a data field morphism.

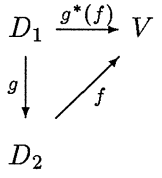
Definition Let $[D_1 \to V]$ and $[D_2 \to V]$ be two data field domains. An index domain morphism $g : D_1 \to D_2$ induces a data field morphism

$$g^* : [D_2 \to V] \to [D_1 \to V]$$

defined by

$$g^*(f) = f \circ g.$$

Example Let $f : D_2 \to V$ be a data field and $g : D_1 \to D_2$ an index domain morphism. Then the following diagram commutes:



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

We use 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

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

is an isomorphism.

Definition The process of converting a function $f : D \times E \to V$ to $\phi(f) : D \to [E \to V]$ is a data field morphism known as *currying*.

3.2 Parallel Programs and Computation Fields

A Crystal program was defined as a set of mutually recursive definitions. The semantics of a Crystal program can be given by the standard fixed point technique.

In Crystal programs, we define the nodes of index domains, but not the arcs. In this section we show how the arcs can be defined from the data dependency that can be extracted from the program text. Next, we combine the data fields defined by each definition into a computation field representing the causal dependency between the data fields. The denotation of a program, consisting of a number of definitions, then is a computation field.

Data Dependency

Consider the following Crystal program:

$$\begin{aligned} f &= \lambda(x) : D_1.\tau_1[f,g], \\ g &= \lambda(x,y) : D_2.\tau_2[f,g]. \end{aligned}$$

Definition An *instance* of a definition is an equation formed by the application of both sides of the definition to some argument. An *instance* of a parallel program is a set of instances of each of its definitions.

Example In the case of the above program, an example of an instance is

$$\{\, f(2) = \tau_1[f,g](2),\ g(2,5) = \tau_2[f,g](2,5) \,\}$$

where 2 is in D_1 and $(2,5)$ is in D_2.

Let f be a function symbol and i be an index in the domain of f. The expression fi will be called an application, and we think of it as indicating a node in the domain of f. We will say the node fi to mean the node i in domain of f. Nested applications denote multiple indices. For example, $g(j, fi)$ denotes the nodes fi, and $g(j,k)$ where k is the value of the function denoted by f at node denoted by i.

Definition Given an instance of a parallel program, the node on the left hand side of each equation is said to be *causally dependent* on the nodes occurring on the right hand side.

Definition Let P be a parallel program defining the data fields $f_i : D_i \to V_i$ for $0 \leq i \leq n$. The *computation field* defined by P is the coproduct

$$[f_0, \ldots, f_n] : \sum\nolimits_J i : 0 \,..\, n \; D_i \to \sum i : 0 \,..\, n \; V_i,$$

where J is the glue consisting of all the new arcs representing the causal dependencies between nodes of different index domains. The communication metric, $\gamma(D_0 +_J \cdots + D_n)$, is induced by the local metrics, $\gamma(D_i)$, of each of the components with new costs defined on the new arcs.

Representing data dependencies of program by directed graphs are widely used [8,14,21].

Proposition 3.1 *A parallel program is well-defined if the domain of its computation field is well-founded.*

Proof: Well-foundedness of the domain implies that there is no infinite chain of data dependencies and no cyclic dependencies. If all the initial values are defined, the whole computation is defined. □

Data fields embody the geometric structure of the data values in spacetime. The computation field of a parallel program embodies the causal dependency of values between the different data fields so that it represent the intention.

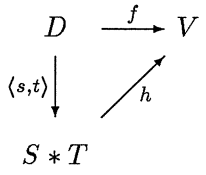
Since computation fields are a subclass of data fields, the notion of computation field morphism is the same as for data fields.

Spacetime Realization

Computation fields embody the logical communication costs, but before a real computation can be carried out, they need to be embedded in a spacetime domain. Spacetime domains are index domains where the arcs and the communication metric are explicitly given. This is in contrast to the index domains of data fields which are determined from the program text.

Definition Let $f : D \to V$ be a computation field. A *spacetime realization* of f is a computation field $h : S * T \to V$, where $S * T$ is a spacetime domain with communication

metric $\gamma(S)$ on S, and an index domain injection $\langle s,t\rangle : D \to S * T$, known as the *spacetime embedding*, making the following diagram commute:



Note that $S * T$ is well-founded for any domain S since T is well-ordered. Also, since domain injections preserve paths, causal dependency of C is preserved in $S * T$.

The metric on D induced by the embedding $\langle s,t\rangle$ gives us precise physical performance measures.

If the domain S has suitable geometry, it makes sense to talk of the minimum area or diameter of S that realizes a computation field f, and the volume of the injection in $S * T$.

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

1. Minimize time. Pick $\langle s,t\rangle$ so as to minimize the length of T.
2. Maximize efficiency. Pick $\langle s,t\rangle$ so as to maximize the ratio of the volume of $\langle s,t\rangle(D)$ inside the minimum bounding volume $S * T$.

3.3 Refinement and Reshaping

An important parallel-program optimization consists in reshaping the data field in order to minimize communication cost for a given space of processors. Domain morphisms embody the exact relationship between two domains. Simple reshaping can be achieved by using *domain isomorphisms*. In general, when there is no isomorphism, we define a pair of morphisms, called *refinement morphism*, that are almost inverses of each other to indicate how the two domains are related.

Since data fields are defined over index domains, these concepts also apply to data fields.

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 $h \circ g = 1_D$. We single out one, called the *conjugate* of g, denoted $g^\bullet$, to be the minimal such morphism under the pointwise ordering of morphisms. The conjugate maps all the nodes that are images of nodes back to their preimage, but is undefined on all the other nodes. Clearly, for any injection g, $(g, g^\bullet)$ is a refinement morphism.

Example Let U and V be interval domains, then (g,h) is a refinement morphism from U to $U \times V$ where 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 $(g,h) : D_1 \to D_2$ be a refinement morphism. The *data field refinement* induced by (g,h), denoted $(g,h)^* : [D_2 \to V] \to [D_1 \to V]$, is a pair of data field morphisms

$$\begin{aligned} g^* &: [D_2 \to V] \to [D_1 \to V], \\ h^* &: [D_1 \to V] \to [D_2 \to V] \end{aligned}$$

such that $f_1 = g^*(f_2) = f_2 \circ g$ and $h^*(f_1) = f_1 \circ h \sqsubseteq f_2$.

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 $D = 0..3$, $E = 0..3$, and $T = 0..6$ be domains. Then,

$$g = \lambda(i,j) : D^2.(i, i+j) : E * T$$

is an affine morphism.

Figure 2 shows the effect of reshaping. The virtical axis is the first component and the horizontal axis is the second component. A spacetime embedding of the domain D^2 requires a time product $D^2 * T$, while the reshaped domain $E * T$ is a spacetime domain already. Hence only a linear number of processors instead of a quadratic number of processors are needed after reshaping.

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 some index domain. Figure 3 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

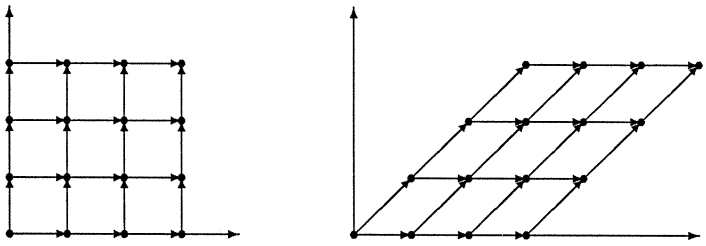


Figure 2: Affine morphism.

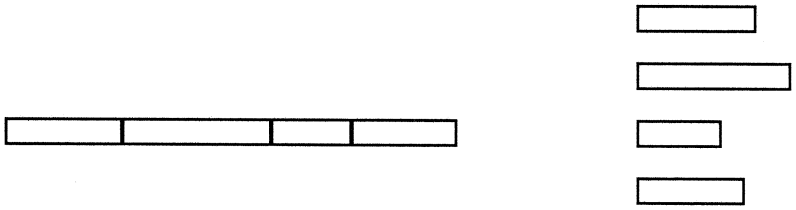


Figure 3: Partition morphism.

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

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

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 4, a contraction resulting in two layers is depicted.

Fan-in and Fan-out Refinements

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

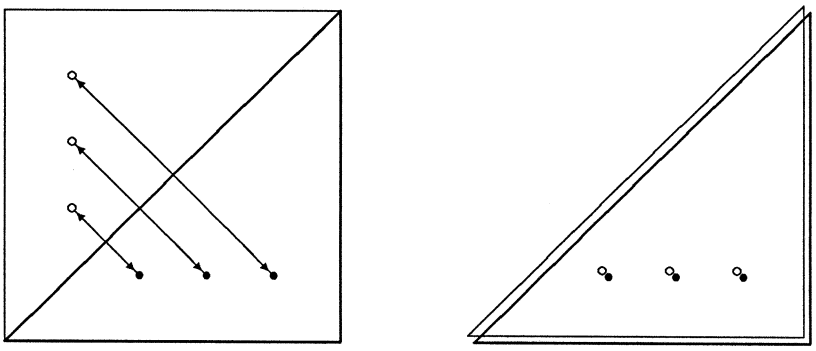


Figure 4: Contraction morphism.

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

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

Figure 5 shows how a large fan-out can be smoothed by a left-associative tree or a balanced tree.

4 A Theory of Crystal

In this section we present an equational theory of the Crystal language. This theory provides the formal foundation for program transformations which realize optimization at the source level. The elements of the equational theory comes from [3,25]. Next, the elements of the Crystal metalanguage is presented in which the program transformations can be expressed.

4.1 An Equational Theory

An equational theory of Crystal consists of the signature Σ of Crystal and a set of equations E.

Signature

The signature Σ consists of the set of sorts S containing int, bool and dom which correspond to the data types of Crystal, and a family of operators Ω of various types over the sorts. For example, $\Omega_{\mathsf{int},\mathsf{int}\to\mathsf{int}}$ is a member of Ω that contain all the operators of type $\mathsf{int}, \mathsf{int} \to \mathsf{int}$, and in particular, will have operators corresponding to the primitive arithmetic functions. Also, we assume variables over all the different sorts.

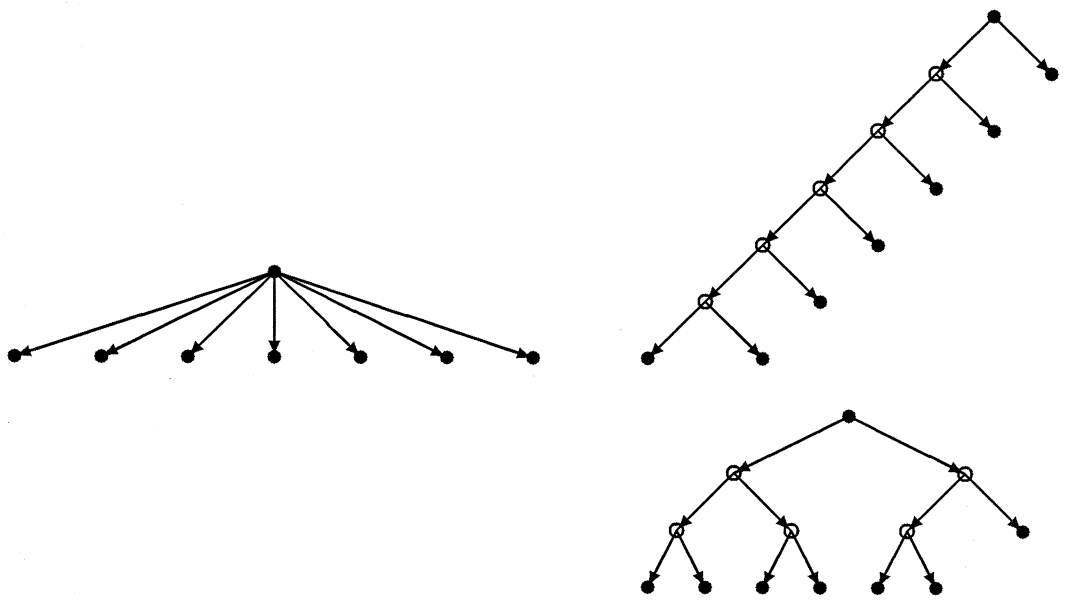


Figure 5: Fan-out refinement.

Equations

An equation is a pair of terms over the signature Σ, that is universally quantified over a set of the free variables that includes the free variables in the two terms. Terms are well-formed expressions in Crystal. The set of equations, E, contains all the equations that are true of the language Crystal interpreted in a standard model. In particular E satisfies the following conditions.

1. Contains all equations derived from the α, β and η conversion rules of the λ-calculus suitably translated into Crystal:

$$\begin{aligned} \lambda x.M &= \lambda y.M[x/y] \\ (\lambda x.M)N &= M[x/N] \\ M &= \lambda x.M(x) \end{aligned}$$

   where $M[x/N]$ denotes the term where all free occurrences of x are replaced by N. The parameters may have type information, in which case they need to be suitably modified.

2. Closed under reflexivity, symmetry and transitivity of the equality relation:

$$\begin{aligned} M &= M \\ M = N &\Rightarrow N = M \\ M = L \text{ and } L = N &\Rightarrow M = N \end{aligned}$$

3. Closed under substitution of terms for equals:

$$M = N \Rightarrow C[M] = C[N]$$
$$M = N \Rightarrow M[H/K] = N[H/K]$$

where $C[\cdot]$ is a term except for one missing subterm, H is a subterm occurring in both M and N, and $M[H/K]$ denotes the substitution of the subterm H by K.

4. Contains equations involving various functions. Let $[d_1, d_2]$ be a coproduct function with injections ι_1 and ι_2, f and h_i functions, e_i be expressions, and b_i be boolean expressions, then E contains the following.

   coproduct

$$d_1 = [d_1, d_2] \circ \iota_1 \quad \text{and} \quad d_2 = [d_1, d_2] \circ \iota_2$$

   dist-comp-coprod

$$f \circ [d_1, d_2] = [f \circ d_1, f \circ d_2]$$

   dist-comp-abs

$$f \circ \left(\lambda x. \begin{Bmatrix} b_1 \to h_1(x) \\ b_2 \to h_2(x) \end{Bmatrix}\right) = \lambda x. \begin{Bmatrix} b_1 \to f \circ h_1(x) \\ b_2 \to f \circ h_2(x) \end{Bmatrix}$$

   dist-app-if

$$f(\begin{Bmatrix} b_1 \to e_1 \\ b_2 \to e_2 \end{Bmatrix}) = \begin{Bmatrix} b_1 \to f(e_1) \\ b_2 \to f(e_2) \end{Bmatrix}$$

Definitions as Equations

A definition in Crystal introduces a new operator symbol to the family of operators Ω, and adds the defining equation to the set of equations E in the theory of Crystal.

Consider the definition

$$f = \lambda x.\tau[f],$$

where $\tau[f]$ is a term (possibly containing f) of Crystal. It defines a function to be denoted by f. In the language Crystal the equality symbol is used for definitions, but once the function has been defined, then we can consider f as satisfying the equation.

Definition A *definition* in the theory of Crystal is a special kind of equation where the left hand side is a variable and the right hand side is some expression.

In the framework of [3], a definition enriches a theory with a new operator and a new equation. For a mutually recursive set of function definitions, we consider the functions that are implicitly defined as satisfying all the defining equations in the enriched theory.

4.2 The Metalanguage

The metalanguage will be used to formalize transformations of programs written in Crystal. Like any other metalanguage, the Crystal metalanguage consists of basic constructors and selectors for each of the constructs in Crystal and operations that manipulate programs. The metalanguage borrows ideas from ML of Edinburgh LCF [9] and Brian Smith's 3-Lisp[27].

Constructors, Selectors, and Predicates

For each construct of the object language, there exists a metalanguage constructor that takes the components and produces the construct. For example, for the conditional expression, the constructor takes a list of guarded expressions and returns the conditional expression made up of the given guarded expressions.

We will use Greek letters to denote the metavariables ranging over the expressions of Crystal. The quine quasi quote are used to name the expressions of Crystal which may contain metavariables.

The following is a list of the basic constructors, selectors. The expressions in quine quotes are the concrete form of the program constructed. For each constructor mk-name the predicate that tests it has the form name?.

Equations

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

Lambda abstraction

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

Eta Abstraction

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

Application

$$\begin{aligned} \text{mk-app}(\tau_1, \tau_2) &\equiv \ulcorner \tau_1 \tau_2 \urcorner \\ \text{rator}(\text{mk-app}(\tau_1, \tau_2)) &= \tau_1 \\ \text{rand}(\text{mk-app}(\tau_1, \tau_2)) &= \tau_2 \end{aligned}$$

Pairs and Triples

$$\begin{aligned} \mathsf{mk\text{-}pair}(\tau_1, \tau_2) &\equiv \ulcorner(\tau_1, \tau_2)\urcorner \\ \mathsf{mk\text{-}triple}(\tau_1, \tau_2, \tau_3) &\equiv \ulcorner(\tau_1, \tau_2, \tau_3)\urcorner \end{aligned}$$

The selectors are first, second, and third and they pick out the respective components.

Guarded Expression

$$\begin{aligned} \mathsf{mk\text{-}gexp}(\tau_1, \tau_2) &\equiv \ulcorner \tau_1 \rightarrow \tau_2 \urcorner \\ \mathsf{guard}(\mathsf{mk\text{-}gexp}(\tau_1, \tau_2)) &= \tau_1 \\ \mathsf{exp}(\mathsf{mk\text{-}gexp}(\tau_1, \tau_2)) &= \tau_2 \end{aligned}$$

where τ_1 is a boolean expression and τ_2 is any expression.

Conditional

$$\begin{aligned} \mathsf{mk\text{-}if}(\tau_1, \tau_2) &\equiv \ulcorner \left\{ \begin{array}{c} \tau_1 \\ \tau_2 \end{array} \right\} \urcorner \\ \mathsf{first\text{-}gexp}(\mathsf{mk\text{-}if}(\tau_1, \tau_2)) &= \tau_1 \\ \mathsf{second\text{-}gexp}(\mathsf{mk\text{-}if}(\tau_1, \tau_2)) &= \tau_2 \end{aligned}$$

Coproduct

$$\mathsf{mk\text{-}coprod}(\tau_1, \tau_2) \equiv \ulcorner[\tau_1, \tau_2]\urcorner$$

The selectors are first and second, and they pick out the respective components. In general there are coproducts for arbitrary number of functions.

We will not explicitly define constructors for boolean expressions and arithmetic expressions. They can be constructed as applications of appropriate functions over pairs or triples of constants or variables or other expressions, inductively.

Operators

The following is a list of operations that are needed for program transformation. The unfold and fold are taken from [2] and are made into operators that take an equation and a function name or a term as arguments.

Let κ range over equations, τ over terms, and ϕ over function names.

$\mathsf{expand}(\kappa, \phi)$ Substitutes all occurrence of ϕ in κ with the right hand side of the equation defining ϕ.

$\mathsf{reduce}(\tau)$ β-reduces the β-redex τ.

$\mathsf{normalize}(\kappa)$ β-reduces all β-redexes occurring in κ.

unfold(κ, ϕ) replaces the function named by ϕ and its argument with its body with appropriate substitutions of the arguments in the equation κ. Note that this is equivalent to **expand** of the function followed by a **reduce** or **normalize**.

fold(κ, ϕ) is the inverse of **unfold** which replaces a term by an equivalent function applied to suitable arguments.

subst(κ, τ_1, τ_2) substitutes the subterm τ_1 with τ_2 in the equation κ, which is only done when $\tau_1 = \tau_2$ is in the theory.

simplify-arith(κ) simplifies the arithmetic and boolean expressions into some canonical form.

simplify-comp(κ) simplifies the composition of functions and eliminate identities.

right-comp(κ, τ) composes the function denoted by τ on the right to both sides of the equation κ.

dist-comp-abs(κ, ϕ) Distribute the function ϕ over abstraction and compose it with the function of the body.

dist-app-if(κ, ϕ) Distribute the function ϕ over the application of a conditional and apply it to the expressions.

dist-comp-coprod(κ, ϕ) Left distribute the function ϕ composed with a coproduct over the coproduct.

def(ϕ) Denotes the equation that defines ϕ.

reshape(**def**($\ulcorner a \urcorner$), $\ulcorner g \urcorner$, $\ulcorner b \urcorner$) Composite operator that takes the 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. A concrete example will be given below.

Using these operators and the constructors and selectors defined above, we can formalize the informal program transformation. The metalanguage provides the facility for constructing and manipulating Crystal programs. The operators are defined to be consistent with the equational theory of Crystal. The equational theory contains the algebraic identities of program expressions and the inference rules for equality and lambda expressions based on the conversions rules of Church's lambda calculus [1]. The α and β rules are well understood, but we show that η-abstraction is needed in order to simplify function expressions.

5 Data Field Morphisms and Program Optimizations

A Crystal program consists of a set of definitions. Each definition defines a data field and the program defines a computation field. The computation field contains not only the extension of the program as a set of functions, but also the intentional information contained in the data dependency between the indices in the various domains.

The task of optimization for Crystal program consists in analysing the communication metric of the underlying index domains, and deriving a reshape or refinement morphism

which will decrease some measure of complexity, for example, total computation time, or resource utilization, then transforming the program according to the equational theory so that the new denotation satisfies the reshape or refinement morphism.

For reshape morphisms, the transformation is straight forward. For refinement morphisms, extra program text must be added to deal with the additional nodes of the refined domain. This section describes the strategy for these morphisms and their corresponding program transformations.

5.1 Reshape Morphism

For simplicity, consider a program consisting of one definition:

$$a = \lambda x : D.e_1[a], \tag{1}$$

where $e[a]$ is an expression in x possibly containing a. By an intentional abuse of notation, let a also be the data field denoted by this program.

Next, let the reshape morphism g and its inverse be defined by

$$\begin{aligned} g &= \lambda x : D.e_2 : E, \\ g^{-1} &= \lambda y : E.e_3 : D. \end{aligned}$$

Techniques for finding reshape morphisms for restrictive classes is known in the literature, see for example [20,19,24,16,5,13,17,18].

Functionally, what we want is a data field $\hat{a}$ satisfying

$$\hat{a} = a \circ g^{-1}.$$

However, merely executing g^{-1} and then a does not decrease the communication costs. What we want is a new definition of $\hat{a}$ which does not contain a or g.

Using the equations of the theory, we outline the strategy for obtaining a new definition for $\hat{a}$ from the definitions of a and g.

1. First, using the identity
$$a = \hat{a} \circ g$$
substitute all occurrences of a in 1.

2. Next, using a combination of unfoldings of g and g^{-1} and various other equational transformations given in the theory, eliminate all occurrences of g and g^{-1}.

The first example in the following section gives a step by step transformation formalized in the metalanguage. The second example deals with a domain contraction which requires a couple of extra steps to deal with the coproduct arrow.

5.2 Refinement Morphism

For refinement morphisms, the program transformation cannot be carried out in a purely deterministic way since the refinement maps the original domain into another domain with extra nodes and arcs. The prgrammer or the programming environment must indicate the type of transformation that is required.

In this section we present the transformations that are needed for the fan-in and fan-out refinement morphisms.

Fan-in Reduction

Consider the program

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

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

The reduction can be done in numerous ways. For each of the three natural orders of reductions, the left associative reduction, the right associative reduction, and the balanced tree associative reductions, denoted by $\setminus_L$, $\setminus_R$, and $\setminus_B$, the definition of a will be different. For the left-associative reduction, the program for c becomes

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

where

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

Note that the square brackets denote syntactic substitution. 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 $\textsf{fan-in-red}(\textsf{def}(\ulcorner c \urcorner), \ulcorner Z_L \urcorner, \ulcorner a \urcorner)$.

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.

By implementing a left-associative tree as a linear domain, Z_L can be simplified to the following:

$$Z_L^0[a, D, H, N_0, f] \;\equiv\; \lambda l : D.\lambda k : N_0. \left\{ \begin{array}{l} k = 0 \to \mathrm{id}(f) \\ k > 0 \to f(a(l)(k-1), H(l)(k)) \end{array} \right\},$$

where $N_0 = \{ 0 \} \cup N$.

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 $\text{tree}_B(D, r)$ 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

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

6 Examples of Optimizations

In this section we present detailed examples of the formal transformation of programs using the metalanguage operators.

6.1 Affine Domain Morphism

The Program

Consider the following program which uses the third dimension in a and b to copy the matix values in order to reduce hot spots:

$$
\begin{aligned}
D &= 0\,..\,n,\\
E &= 0\,..\,2n-1,\\
T &= 0\,..\,3n-2,\\
a &= \lambda(i,j,k):D^3.\left\{\begin{array}{l} j=0 \to A_{ik} \\ 0<j\le n \to a(i,j-1,k)\end{array}\right\}\\
b &= \lambda(i,j,k):D^3.\left\{\begin{array}{l} i=0 \to B_{kj} \\ 0<i\le n \to b(i-1,j,k)\end{array}\right\}\\
c &= \lambda(i,j,k):D^3.\left\{\begin{array}{l} k=0 \to 0 \\ 0<k\le n \to a(i,j,k)\times b(i,j,k)+c(i,j,k-1)\end{array}\right\}
\end{aligned}
$$

A direct realization of this algorithm in a spacetime domain will result in injecting D^3 into $D^3 * T_n$ which due to the causal dependency will be very sparse, resulting in many idle processors.

Since the communication is only along each coordinate by 1, D has the Manhattan metric. The above program defines the data fields a, b, and c which are the components of the computation field

$$[a,b,c]: D +_J D + D \to V,$$

where the glue J consists of arcs connecting the nodes from the first two summand to the corresponding node in the third summand. The communication metric will be the Manhattan metric on each component of the coproduct domain while the inter-component cost is defined to be 1 between corresponding elements, i.e., $\gamma(D+D+D)(\iota_i(x),\iota_j(x)) = 1$ for each x in D and i, j from 1 to 3.

Suppose we are given the following affine morphism:

$$
\begin{aligned}
g &= \lambda(i,j,k):D^3.(i+k, j+k, i+j+k):(E\times E)*T\\
g^{-1} &= \lambda(i,j,k):E^2*T.(k-j,k-i,i+j-k):D^3
\end{aligned}
$$

Figure 6 shows how the two domains are related. In the naive realization, the domain D^3 is replicated in time, but the affine morphism allows a spacetime realization with two space coordinates and a time coordinate.

The Derivation of an Optimized Program

Given the definition of a data field a and a affine domain morphism g, our goal is to derive the definition of another data field $\hat{a}$ such that $a = \hat{a} \circ g$. The derivations for the other definitions are identical.

Let κ_0 denote the equation defining a above. The sequence of operations is as follows:

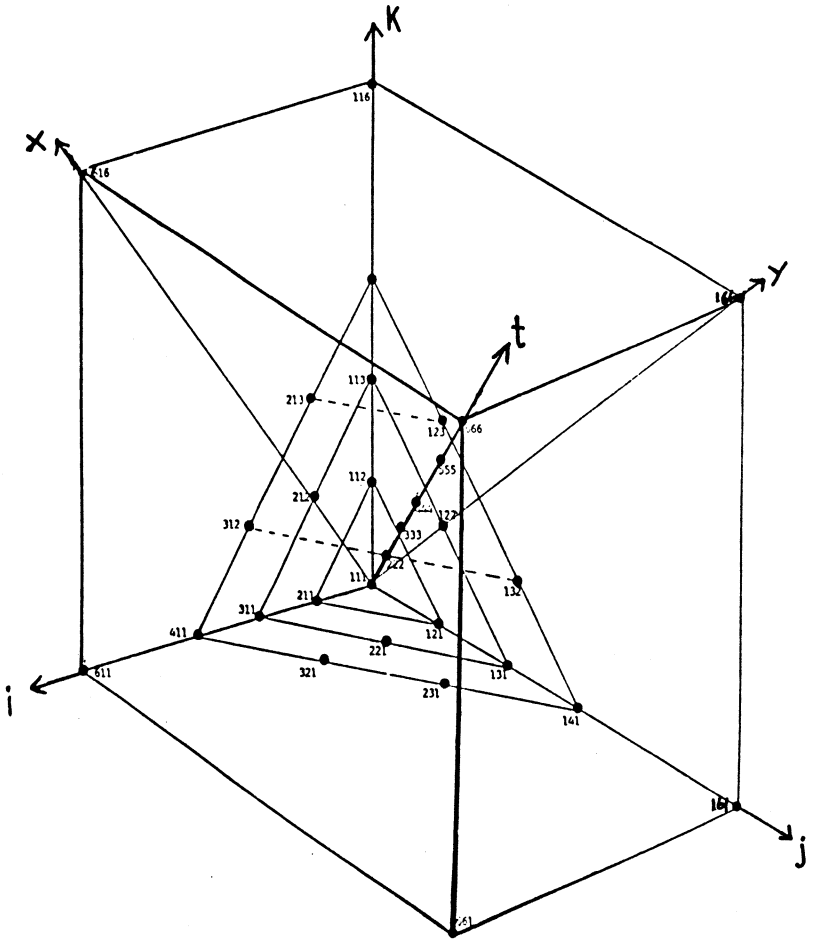


Figure 6: An Affine Domain Morphism.

1. Substitute $\ulcorner \hat{a} \circ g \urcorner$ for $\ulcorner a \urcorner$ in κ_0. $\kappa_1 \equiv \mathsf{subst}(\kappa_0, \ulcorner a \urcorner, \ulcorner \hat{a} \circ g \urcorner)$:

$$\hat{a} \circ g = \lambda(i,j,k) : D^3 . \left\{ \begin{array}{l} j = 0 \rightarrow A(i,k) \\ 0 < j <= n \rightarrow \hat{a} \circ g(i, j-1, k) \end{array} \right\}$$

2. Right compose both sides of the equation with g^{-1}, the inverse of g. $\kappa_2 \equiv \mathsf{right\text{-}comp}(\kappa_1, \ulcorner g^{-1} \urcorner)$:

$$\hat{a} \circ g \circ g^{-1} = \left[\lambda(i,j,k) : D^3 . \left\{ \begin{array}{l} j = 0 \rightarrow A(i,k) \\ 0 < j \leq n \rightarrow \hat{a} \circ g(i, j-1, k) \end{array} \right\} \right] \circ g^{-1}$$

3. Simplify composition of g and g^{-1}). $\kappa_3 \equiv \mathsf{simplify\text{-}comp}(\kappa_2)$:

$$\hat{a} = \left[\lambda(i,j,k) : D^3 . \left\{ \begin{array}{l} j = 0 \rightarrow A(i,k) \\ 0 < j \leq n \rightarrow \hat{a} \circ g(i, j-1, k) \end{array} \right\} \right] \circ g^{-1}$$

4. Expand g. $\kappa_4 \equiv \mathsf{expand}(\kappa_3, \ulcorner g \urcorner)$:

$$\hat{a} = \Big[\lambda(i,j,k) : D^3 . \begin{cases} j = 0 \to A(i,k) \\ 0 < j \leq n \to \hat{a} \circ \\ \quad \Big(\lambda(i,j,k) : D^3 . (i+k, j+k, i+j+k)\Big)(i, j-1, k) \end{cases}\Big] \circ g^{-1}$$

5. Distribute composition over abstraction. $\kappa_5 \equiv \mathsf{dist\text{-}comp\text{-}abs}(\kappa_4, \ulcorner \hat{a} \urcorner)$:

$$\hat{a} = \Big[\lambda(i,j,k) : D^3 . \begin{cases} j = 0 \to A(i,k) \\ 0 < j \leq n \to \\ \quad \Big(\lambda(i,j,k) : D^3 . \hat{a}(i+k, j+k, i+j+k)\Big)(i, j-1, k) \end{cases}\Big] \circ g^{-1}$$

6. Normalize. $\kappa_6 \equiv \mathsf{normalize}(\kappa_5)$:

$$\hat{a} = \Big[\lambda(i,j,k) : D^3 . \begin{cases} j = 0 \to A(i,k) \\ 0 < j \leq n \to \hat{a}(i+k, j-1+k, i+j-1+k) \end{cases}\Big] \circ g^{-1}$$

7. Eta abstract the body of the right hand side.
$\kappa_7 \equiv \mathsf{mk\text{-}eqn}(\mathsf{lhs}(\kappa_6), \mathsf{mk\text{-}eta\text{-}abs}(\ulcorner (i,j,k) : E^2 * T \urcorner, \mathsf{rhs}(\kappa_6)))$:

$$\begin{aligned} \hat{a} \;=\; & \lambda(i,j,k) : E^2 * T . \Big[\Big(\lambda(i,j,k) : D^3 . \\ & \begin{cases} j = 0 \to A(i,k) \\ 0 < j \leq n \to \\ \quad \hat{a}(i+k, j-1+k, i+j-1+k) \end{cases}\Big) \circ g^{-1}\Big](i,j,k) \end{aligned}$$

8. Unfold the composition and normalize. $\kappa_8 \equiv \mathsf{normalize}(\mathsf{unfold}(\kappa_7, \ulcorner \circ \urcorner))$:

$$\hat{a} = \lambda(i,j,k) : E^2 * T . \Big(\lambda(i,j,k) : D^3 . \begin{cases} j = 0 \to A(i,k) \\ 0 < j \leq n \to \\ \quad \hat{a}(i+k, j-1+k, i+j-1+k) \end{cases}\Big)(g^{-1}(i,j,k))$$

9. Unfold g^{-1}. $\kappa_9 \equiv \mathsf{unfold}(\kappa_8, \ulcorner g^{-1} \urcorner)$:

$$\begin{aligned} \hat{a} \;=\; & \lambda(i,j,k) : E^2 * T . \Big(\lambda(i,j,k) : D^3 . \\ & \begin{cases} j = 0 \to A(i,k) \\ 0 < j \leq n \to \\ \quad \hat{a}(i+k, j-1+k, i+j-1+k) \end{cases}\Big)(k-j, k-i, i+j-k)) \end{aligned}$$

10. Normalize. $\kappa_{10} \equiv \mathsf{normalize}(\kappa_9)$:

$$\hat{a} = \lambda(i,j,k) : E^2 * T. \left\{ \begin{array}{l} k - i = 0 \rightarrow A(k-j, i+j-k) \\ 0 < k - i \leq n \rightarrow \\ \quad \hat{a}(k-j+i+j-k, k-i+i+j-k-1, \\ \quad\quad k-j+k-i-1+i+j-k) \end{array} \right\}$$

11. Simplify the arithmetic. $\kappa_{11} \equiv \mathsf{simplify\text{-}arith}(\kappa_{10})$:

$$\hat{a} = \lambda(i,j,k) : E^2 * T. \left\{ \begin{array}{l} k - i = 0 \rightarrow A(k-j, i+j-k) \\ 0 < k - i \leq n \rightarrow \hat{a}(i, j-1, k-1) \end{array} \right\}$$

The final equation is in the form of a definition, without a or g appearing in the expression.

6.2 Domain Contraction

In this section we illustrate the derivation of the layers of a domain contraction using a linear programming example.

The Program

Let $m(i,j) \equiv \lceil \frac{j+i}{2} \rceil$ and let h_1 and h_2 be expressions.

$$\begin{aligned} D &= \{\, (i,j,k) \mid 1 \leq i \leq k \leq j \leq n \,\}, \\ E &= \{\, (i,j,k) \mid (i,j,k) \in D \wedge k \geq m(i,j) \,\}, \\ a &= \lambda(i,j,k) : D. \left\{ \begin{array}{l} j = k \rightarrow c(i,j,k) \\ j > k \rightarrow a(i, j-1, k) \end{array} \right\} \\ b &= \lambda(i,j,k) : D. \left\{ \begin{array}{l} i = k \rightarrow c(i,j,k) \\ i < k \rightarrow b(i+1, j, k) \end{array} \right\} \\ c &= \lambda(i,j,k) : D. \left\{ \begin{array}{l} j - i = 1 \rightarrow C_i \\ j - i > 1 \rightarrow \\ \quad \left\{ \begin{array}{l} k = m(i,j) \rightarrow h_1[a(i,j,k), b(i,j,k), \\ \quad\quad a(i,j,i+j-k), b(i,j,i+j-k)] \\ m(i,j) < k < j \rightarrow \\ \quad\quad h_2[c(i,j,k-1), a(i,j,k), b(i,j,k), \\ \quad\quad\quad a(i,j,i+j-k), b(i,j,i+j-k)] \\ k = j \rightarrow c(i,j,k-1) \end{array} \right\} \end{array} \right\} \end{aligned}$$

Just like in the previous example, let D have the Manhattan metric. The above program defines the data fields a, b, and c and the corresponding computation field $[a,b,c] : D +_J$

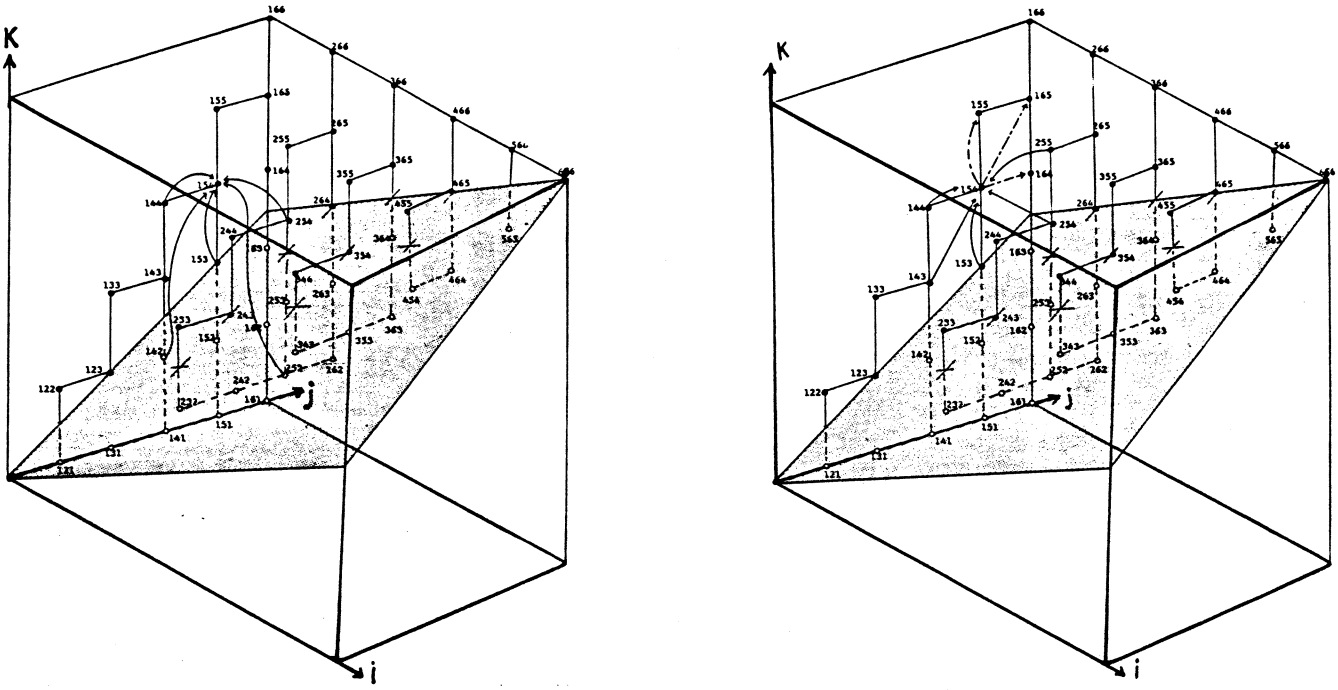


Figure 7: A domain contraction.

$D + D \rightarrow V$ with the Manhattan metric on each component of the domain and the inter-component cost defined to be zero between corresponding elements:

$$\gamma(D +_J D + D)(\iota_i(x), \iota_j(x)) = 0,$$

for each x in D and i, j from 1 to 3.

Notice that the causal dependency in the definition for c requires that the left-hand-side node depends on a right-hand-side node that is distance $k - (i + j - k)$ away.

Analysis of the computation field indicates a way to reduce the long distance communication by reflecting one half of the domain into the other. This defines a contraction morphism g and its inverse g^{-1}:

$$g = \lambda(i,j,k) : D. \left\{ \begin{array}{l} k \geq m(i,j) \rightarrow \iota_1(i,j,k) \\ k < m(i,j) \rightarrow \iota_2(i,j,i+j-k) \end{array} \right\} : E + E$$

$$g^{-1} \equiv [l_1, l_2] = [\lambda(i,j,k) : E.(i,j,k), \lambda(i,j,k) : E.(i,j,i+j-k)] : D$$

The effect of this contraction is to fold the computation field upon itself so that the nodes requiring communication over distance $k-(i+j-k)$ can now be made to have a small constant communication cost. The Figure 7 illustrates the before and after of the contraction. The shaded plane represents the plane of reflection and all the nodes below it are reflected into the upper part of the domain. The long distance communication in the first diagram become constant distance communication.

The Derivation of an Optimized Program

The derivation will only be carried out for the definition of a. The derivations for b and c are identical.

From the definition of a and the definition of the contraction, our goal is to derive the definition of the layers of the contraction d_1 and d_2 such that $a = [d_1, d_2] \circ g$, where $[d_1, d_2]$ denotes the coproduct arrow of the coproduct construction. The procedure is essentially the same as for the affine domain morphism.

1. Substitute $\ulcorner [d_1, d_2] \circ g \urcorner$ for $\ulcorner a \urcorner$ in κ_0.
   $\kappa_1 \equiv \mathsf{subst}(\kappa_0, \ulcorner a \urcorner, \ulcorner [d_1, d_2] \circ g \urcorner)$:

$$[d_1, d_2] \circ g = \lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to [d_1, d_2] \circ g(i, j-1, k) \end{cases}$$

2. Compose both sides of the equation on the right with $[l_1, l_2]$, the inverse of g.
   $\kappa_2 \equiv \mathsf{right\text{-}comp}(\kappa_1, \ulcorner [l_1, l_2] \urcorner)$:

$$[d_1, d_2] \circ g \circ [l_1, l_2] = \left[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to [d_1, d_2] \circ g(i, j-1, k) \end{cases}\right] \circ [l_1, l_2]$$

3. Simplify composition of g and $[l_1, l_2]$. $\kappa_3 \equiv \mathsf{simplify\text{-}comp}(\kappa_2)$:

$$[d_1, d_2] = \left[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to [d_1, d_2] \circ g(i, j-1, k) \end{cases}\right] \circ [l_1, l_2]$$

4. Expand g. $\kappa_4 \equiv \mathsf{expand}(\kappa_3, \ulcorner g \urcorner)$:

$$[d_1, d_2] = \left[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to [d_1, d_2] \circ \\ \quad \left(\lambda(i,j,k) : D. \begin{cases} k \geq m(i,j) \to \\ \quad \iota_1(i,j,k) \\ k < m(i,j) \to \\ \quad \iota_2(i,j,i+j-k) \end{cases}\right)(i, j-1, k) \end{cases}\right] \circ [l_1, l_2]$$

5. Distribute composition over abstraction. $\kappa_5 \equiv \mathsf{dist\text{-}comp\text{-}abs}(\kappa_4, \ulcorner [d_1, d_2] \urcorner)$:

$$[d_1, d_2] = \left[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to \left(\lambda(i,j,k) : D. \begin{cases} k \geq m(i,j) \to \\ \quad [d_1, d_2] \circ \iota_1(i,j,k) \\ k < m(i,j) \to \\ \quad [d_1, d_2] \circ \iota_2(i,j,i+j-k) \end{cases}\right)(i, j-1, k) \end{cases}\right] \circ [l_1, l_2]$$

6. Normalize. $\kappa_6 \equiv \mathsf{normalize}(\kappa_4)$:

$$[d_1, d_2] = \Big[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad [d_1, d_2] \circ \iota_1(i,j-1,k) \\ k < m(i,j-1) \to \\ \quad [d_1, d_2] \circ \iota_2(i,j-1,i+j-1-k) \end{cases} \end{cases} \Big] \circ [l_1, l_2]$$

7. Substitute d_1 for $[d_1, d_2] \circ \iota_1$ and d_2 for $[d_1, d_2] \circ \iota_2$.
$\kappa_7 \equiv \mathsf{subst}(\mathsf{subst}(\kappa_6, \ulcorner [d_1, d_2] \circ \iota_1 \urcorner, \ulcorner d_1 \urcorner), \ulcorner [d_1, d_2] \circ \iota_2 \urcorner, \ulcorner d_2 \urcorner)$:

$$[d_1, d_2] = \Big[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases} \Big] \circ [l_1, l_2]$$

8. Left distribute composition over coproduct map ($\tau \circ [l_1, l_2] = [\tau \circ l_1, \tau \circ l_2]$).
$\kappa_8 \equiv \mathsf{dist\text{-}comp\text{-}coprod}(\kappa_7, \ulcorner [l_1, l_2] \urcorner)$:

$$[d_1, d_2] = \Big[\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases} \Big] \circ l_1,$$

$$\lambda(i,j,k) : D. \begin{cases} j = k \to c(i,j,k) \\ j > k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases} \Big] \circ l_2 \Big]$$

9. Eta abstract the two components of the coproduct function.
$\kappa_9 \equiv \mathsf{mk\text{-}coprod}(\mathsf{mk\text{-}eta}(\ulcorner (i,j,k) : E \urcorner, \mathsf{first}(\mathsf{rhs}(\kappa_8))),$
$\mathsf{mk\text{-}eta}(\ulcorner (i,j,k) : E \urcorner, \mathsf{second}(\mathsf{rhs}(\kappa_8))))$:

$$[d_1, d_2] = \Big[\lambda(i,j,k) : E.$$

$$\Big(\lambda(i,j,k):D.\begin{cases} j=k \to c(i,j,k) \\ j>k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases} \circ l_1\Big)(i,j,k),$$

$$\lambda(i,j,k):E.$$

$$\Big(\lambda(i,j,k):D.\begin{cases} j=k \to c(i,j,k) \\ j>k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases} \circ l_2\Big)(i,j,k)\Big]$$

10. Unfold the composition functional and normalize. $\kappa_{10} \equiv \mathsf{normalize}(\mathsf{unfold}(\kappa_9, \ulcorner \circ \urcorner))$:

$$[d_1,d_2] = \Big[\lambda(i,j,k):E.$$

$$\Big(\lambda(i,j,k):D.\begin{cases} j=k \to c(i,j,k) \\ j>k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases}\Big)(l_1(i,j,k)),$$

$$\lambda(i,j,k):E.$$

$$\Big(\lambda(i,j,k):D.\begin{cases} j=k \to c(i,j,k) \\ j>k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases}\Big)(l_2(i,j,k))\Big]$$

11. Unfold l_1 and l_2. $\kappa_{11} \equiv \mathsf{unfold}(\mathsf{unfold}(\kappa_{10}, \ulcorner l_1 \urcorner), \ulcorner l_2 \urcorner)$

$$[d_1,d_2] = \Big[\lambda(i,j,k):E.$$

$$\Big(\lambda(i,j,k):D.\begin{cases} j=k \to c(i,j,k) \\ j>k \to \begin{cases} k \geq m(i,j-1) \to \\ \quad d_1(i,j-1,k)) \\ k < m(i,j-1) \to \\ \quad d_2(i,j-1,i+j-1-k)) \end{cases} \end{cases}\Big)(i,j,k),$$

$$\lambda(i,j,k):E.$$
$$\Big(\lambda(i,j,k):D.\left\{\begin{array}{l} j=k\to c(i,j,k)\\ j>k\to\\ \quad\left\{\begin{array}{l} k\geq m(i,j-1)\to\\ \quad d_1(i,j-1,k))\\ k<m(i,j-1)\to\\ \quad d_2(i,j-1,i+j-1-k))\end{array}\right.\end{array}\right\}\Big)(i,j,i+j-k)\Big]$$

12. Beta-reduce the beta-redexes. $\kappa_{12} \equiv \mathsf{normalize}(\kappa_{11})$:

$$[d_1,d_2] = \Big[\lambda(i,j,k):E.\left\{\begin{array}{l} j=k\to c(i,j,k)\\ j>k\to\left\{\begin{array}{l} k\geq m(i,j-1)\to\\ \quad d_1(i,j-1,k))\\ k<m(i,j-1)\to\\ \quad d_2(i,j-1,i+j-1-k))\end{array}\right.\end{array}\right\},$$
$$\lambda(i,j,k):E.\left\{\begin{array}{l} j=i+j-k\to c(i,j,i+j-k)\\ j>i+j-k\to\\ \quad\left\{\begin{array}{l} i+j-k\geq m(i,j-1)\to\\ \quad d_1(i,j-1,i+j-k))\\ i+j-k<m(i,j-1)\to\\ \quad d_2(i,j-1,i+j-1-(i+j-k)))\end{array}\right.\end{array}\right\}\Big]$$

13. Simplify the arithmetic. $\kappa_{13} \equiv \mathsf{simplify\text{-}arith}(\kappa_{12})$:

$$[d_1,d_2] = \Big[\lambda(i,j,k):E.\left\{\begin{array}{l} j=k\to c(i,j,k)\\ j>k\to\left\{\begin{array}{l} k\geq m(i,j-1)\to\\ \quad d_1(i,j-1,k))\\ k<m(i,j-1)\to\\ \quad d_2(i,j-1,i+j-1-k))\end{array}\right.\end{array}\right\},$$
$$\lambda(i,j,k):E.\left\{\begin{array}{l} k=i\to c(i,j,i+j-k)\\ k>i\to\left\{\begin{array}{l} i+j-k\geq m(i,j-1)\to\\ \quad d_1(i,j-1,i+j-k))\\ i+j-k<m(i,j-1)\to\\ \quad d_2(i,j-1,k-1)))\end{array}\right.\end{array}\right\}\Big]$$

The two derivation sequences are almost identical. In the domain contraction, we have two extra operations that are needed to deal specifically with the coproduct map. In step 7 we use the coproduct diagram to simplify an expression, while in step 8 we distribute a composition over a coproduct.

6.3 Compile-time Optimizations

The Program

As a Crystal program, matrix multiplication is specified as

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

For this exercise, we assume a Manhattan metric on the structure of the target machine.

Fan-in reduction

In the program C, hot spots occur at each index $(i,j) \in D$ since n terms are summed together. By performing fan-in reduction $\mathsf{fan\text{-}in\text{-}red}(\mathsf{def}(\ulcorner C \urcorner), \ulcorner Z_L \urcorner, \ulcorner \widehat{C} \urcorner)$ the new program becomes:

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

Expanding the definition of Z_L^0, we obtain

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

where we have replaced $\mathrm{id}(+)$ with 0.

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 divides a domain into subdomains each of size b or less.

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

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:

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

Let κ_0 denote the equation defining $\widehat{C}$ above. The sequence of operations described in the meta-language operator to obtain the target program after partition morphism is as follows:

1. Substitute $\ulcorner \hat{c} \circ g \urcorner$ for $\ulcorner \widehat{C} \urcorner$ in κ_0. $\kappa_1 \equiv \mathsf{subst}(\kappa_0, \ulcorner \widehat{C} \urcorner, \ulcorner \hat{c} \circ g \urcorner)$:

$$\hat{c} \circ g = \lambda(i,j) : D.\lambda k : N_0.\left\{ \begin{array}{l} k = 0 \to 0, \\ k > 0 \to \hat{c} \circ g(i,j)(k-1) + A(i,k) \times B(k,j) \end{array} \right\}$$

2. Right compose both sides of the equation with g^{-1}, the inverse of g. $\kappa_2 \equiv \mathsf{right\text{-}comp}(\kappa_1, \ulcorner g^{-1} \urcorner)$:

$$\hat{c} \circ g \circ g^{-1} = \left[\lambda(i,j) : D.\lambda k : N_0.\left\{ \begin{array}{l} k = 0 \to 0, \\ k > 0 \to \hat{c} \circ g(i,j)(k-1) + A(i,k) \times B(k,j) \end{array} \right\}\right] \circ g^{-1}$$

3. Simplify composition of g and g^{-1}. $\kappa_3 \equiv \mathsf{simplify\text{-}comp}(\kappa_2)$:

$$\hat{c} = \left[\lambda(i,j) : D.\lambda k : N_0.\left\{ \begin{array}{l} k = 0 \to 0, \\ k > 0 \to \hat{c} \circ g(i,j)(k-1) + A(i,k) \times B(k,j) \end{array} \right\}\right] \circ g^{-1}$$

4. Expand g. $\kappa_4 \equiv \mathsf{expand}(\kappa_3, \ulcorner g \urcorner)$:

$$\begin{aligned} \hat{c} = \Big[&\lambda(i,j) : D.\lambda k : N_0. \\ &\left\{ \begin{array}{l} k = 0 \to 0, \\ k > 0 \to \\ \quad \hat{c} \circ \Big(\lambda(i,j) : D. \\ \quad ((i \operatorname{div} b0, i \bmod b0), (j \operatorname{div} b1, j \bmod b1))\Big)(i,j)(k-1) \\ \qquad +A(i,k) \times B(k,j) \end{array} \right\}\Big] \circ g^{-1} \end{aligned}$$

5. Distribute composition over abstraction. $\kappa_5 \equiv \mathsf{dist\text{-}comp\text{-}abs}(\kappa_4, \ulcorner \hat{c} \urcorner)$:

$$\begin{aligned} \hat{c} = \Big[&\lambda(i,j) : D.\lambda k : N_0. \\ &\left\{ \begin{array}{l} k = 0 \to 0, \\ k > 0 \to \Big(\lambda(i,j) : D. \\ \quad \hat{c}((i \operatorname{div} b0, i \bmod b0), (j \operatorname{div} b1, j \bmod b1))\Big)(i,j)(k-1) \\ \qquad +A(i,k) \times B(k,j) \end{array} \right\}\Big] \circ g^{-1} \end{aligned}$$

6. Normalize. $\kappa_6 \equiv \mathsf{normalize}(\kappa_5)$:

$$\hat{c} = \Big[\lambda(i,j):D.\lambda k:N_0. \left\{\begin{array}{l} k=0 \to 0, \\ k>0 \to \hat{c}((i \operatorname{div} b0, i \bmod b0),(j \operatorname{div} b1, j \bmod b1))(k-1) \\ \qquad +A(i,k)\times B(k,j) \end{array}\right\}\Big] \circ g^{-1}$$

7. Eta abstract the body of the right hand side.
$\kappa_7 \equiv \mathsf{mk\text{-}eqn}(\mathsf{lhs}(\kappa_6), \mathsf{mk\text{-}eta\text{-}abs}(\ulcorner((u0,v0),(u1,v1)):(U_{b0}\times V_{b0})\times(U_{b1}\times V_{b1})\urcorner, \mathsf{rhs}(\kappa_6)))$:

$$\begin{array}{l} \hat{c} = \lambda((u0,v0),(u1,v1)):(U_{b0}\times V_{b0})\times(U_{b1}\times V_{b1}). \\ \quad \Big[\Big(\lambda(i,j):D.\lambda k:N_0. \left\{\begin{array}{l} k=0 \to 0, \\ k>0 \to \hat{c}((i \operatorname{div} b0, i \bmod b0),(j \operatorname{div} b1, j \bmod b1))(k-1) \\ \quad +A(i,k)\times B(k,j) \end{array}\right\}\Big) \circ g^{-1}\Big] \\ \quad ((u0,v0),(u1,v1)) \end{array}$$

8. Unfold the composition and normalize. $\kappa_8 \equiv \mathsf{normalize}(\mathsf{unfold}(\kappa_7, \ulcorner\circ\urcorner))$:

$$\begin{array}{l} \hat{c} = \lambda((u0,v0),(u1,v1)):(U_{b0}\times V_{b0})\times(U_{b1}\times V_{b1}).\Big(\lambda(i,j):D.\lambda k:N_0. \\ \quad \left\{\begin{array}{l} k=0 \to 0, \\ k>0 \to \hat{c}((i \operatorname{div} b0, i \bmod b0),(j \operatorname{div} b1, j \bmod b1))(k-1) \\ \qquad +A(i,k)\times B(k,j) \end{array}\right\}\Big) \\ \quad (g^{-1}((u0,v0),(u1,v1))) \end{array}$$

9. Unfold g^{-1}. $\kappa_9 \equiv \mathsf{unfold}(\kappa_8, \ulcorner g^{-1}\urcorner)$:

$$\begin{array}{l} \hat{c} = \lambda((u0,v0),(u1,v1)):(U_{b0}\times V_{b0})\times(U_{b1}\times V_{b1}).\Big(\lambda(i,j):D.\lambda k:N_0. \\ \quad \left\{\begin{array}{l} k=0 \to 0, \\ k>0 \to \hat{c}((i \operatorname{div} b0, i \bmod b0),(j \operatorname{div} b1, j \bmod b1))(k-1) \\ \qquad +A(i,k)\times B(k,j) \end{array}\right\}\Big) \\ \quad (u0\times b0+v0, u1\times b1+u1) \end{array}$$

10. Normalize. $\kappa_{10} \equiv \mathsf{normalize}(\kappa_9)$:

$$\begin{array}{l} \hat{c} = \lambda((u0,v0),(u1,v1)):(U_{b0}\times V_{b0})\times(U_{b1}\times V_{b1}).\lambda k:N_0. \\ \quad \left\{\begin{array}{l} k=0 \to 0, \\ k>0 \to \hat{c}(((u0\times b0+v0) \operatorname{div} b0, (u0\times b0+v0) \bmod b0), \\ \qquad ((u1\times b1+u1) \operatorname{div} b1, (u1\times b1+u1) \bmod b1))(k-1) \\ \qquad\quad +A((u0\times b0+v0),k)\times B(k,(u1\times b1+u1)) \end{array}\right\} \end{array}$$

11. Simplify the arithmetic. $\kappa_{11} \equiv \mathsf{simplify\text{-}arith}(\kappa_{10})$:

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

Currying

We now want to modify the above definition so as to set the stage for making $v0$ and $v1$ indices of sequential loops while keeping the parallel interpretation of indices $u0$ and $u1$. By currying, we redefine $\tilde{c}$ to be $\hat{c}$:

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

The so-called "interchange of loop index" technique in parallelizing compiler is a special case of currying.

Fan-out Reduction

Note that in the above definition, the distribution of the matrix elements of A and B causes hot spots at every index $(u0,u1)$. To remove these hot spots, we perform fan-out reduction. First, another currying step results in

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

Then $\mathsf{fan\text{-}out\text{-}red}(\mathsf{def}(\ulcorner \hat{c}_A \urcorner), \ulcorner X_L \urcorner, \ulcorner a \urcorner)$ gives us

$$\hat{c}_A = \lambda u1 : U_{b1}.\lambda u0 : U_{b0}.\lambda(v0,v1) : V_{b0} \times V_{b1}).\lambda k : N_0.$$
$$\left\{\begin{array}{l} k = 0 \rightarrow 0, \\ k > 0 \rightarrow \hat{c}(u0,u1)(v0,v1)(k-1) \\ \qquad +a(u1)(u0,v0,k) \times B(k,(u1 \times b1 + u1)) \end{array}\right\}$$

$$a = X_L[a, U_1, \lambda(u0,v0,k) : U_0 \times V_0 \times N_0.A((u0 \times b0 + v0),k)].$$

Fan-out reduction for matrix elements of B can be carried out similarly.

7 Conclusion

Parallel computing has the potential of becoming the way of computing. It's success will depend on the advances in hardware design methodology, parallel programming techniques, and parallel systems software—software that can automatically distribute a programming task over multiple processors in such a way that the inter-processor communication cost is kept low enough to achieve high performance—as well as the development of parallel machines with ever higher performance.

Our theory of parallel program optimization is applicable to the design of special-purpose hardware (e.g. systolic arrays [15]), the design of programs for general purpose parallel machines (e.g. [10,29,26]), and systems software for parallel machines. The examples in this paper have been linear, or piece-wise linear morphisms, but the derivation can handle arbitrary graph homomorphisms and injections.

The task of parallel program optimization can be divided into two phases. The creative phase of discovering a data field morphism that will reshape or refine the original data field into a more efficient one, and the mechanical phase of applying the program transformation on the original program to produce the optimized version.

The discovery of good data field morphisms will require analyzing the different spacetime realizations for a given model of computation. The communication metric induced by the spacetime embedding provides the measure of cost for different implementations. An obvious question that presents itself is whether this phase can be automated. We know that for certain classes there are techniques for finding data field morphisms that are optimal or near optimal [20,19,24,16,5,13,17,18], but in general the techniques will most likely be too expensive.

Granted that an intelligent program transformation system or a compiler for parallel machine that can automatically produce the optimal data field morphism may be unrealistic, we envision a system which contains an extensible set of transformation rules that are applied following some default strategy, with the programmer always having the option of choosing the rules that are to be applied or even explicitly defining new transformations in the metalanguage. The system will need a smooth interface between the automatic functions and program directives and also between the metalanguage transformation and compilation systems.

Acknowledgment This work has been supported in part by the Office of Naval Research under Contract No. N00014-86-K-0564.

References

[1] A. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[2] 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.

[3] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *The 5th International Joint Conference on AI*, pages 1045–1058, 1977.

[4] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

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

[6] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[7] James Dugundji. *Topology.* Allyn and Bacon, 1966.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

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

[10] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer - designing an mimd shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, 1983.

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

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

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

[14] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M.J. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

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

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

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

[18] Björn Lisper. *Synthesizing Synchronous Systems by Static Scheduling in Space-time.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, February 1987.

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

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

[21] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), 1986.

[22] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[23] Alberto Pettorossi. *Methodolgies for Transformations and Memoing in Applicative Languages.* PhD thesis, University of Edinburgh Dept. of Computer Science, October 1984.

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

[25] D. T. Sannella. A set-theoretic semantics for clear. *Acta Informatica*, 21:443–472, 1984.

[26] Charles L. Seitz. The cosmic cube. *CACM*, 22–33, January 1985.

[27] Brian Cantwell Smith. *Reflection and Semantics in Lisp.* Technical Report CSLI-84-8, Center for the Study of Language and Information, Stanford University, December 1984. Also appeared in POPL.

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

[29] Hillis W.D. *The Connection Machine.* MIT Press, 1985.