Yale University
Department of Computer Science

Data Field + Dependency = Parallel Program

Marina Chen, Young-il Choo, and Jingke Li

YALEU/DCS/TR-621
April 1988

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

Data Field + Dependency = Parallel Program

Marina Chen Young-il Choo Jingke Li

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

April 11, 1988

Abstract

This paper describes how Crystal — a language based on familiar mathematical notation and lambda calculus — addresses the issues of programmability and performance for parallel supercomputers. What is new about Crystal (or how is it different from existing functional languages) lies in its model of parallel computation and a theory of parallel program optimization. We illustrate the power of our approach with benchmarks of compiled parallel code from Crystal source. The target machines are hypercube multiprocessors with distributed memory, on which it is considered difficult for functional programs to achieve high efficiency.

1 Introduction

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

Superficially, the idea of parallel processing is simple: allow many processors to work together in order to solve a particular problem. Thus, the length of time it takes to perform the task will be reduced. All sounds well and good, but in order to use parallel machines effectively, many problems must be overcome. First of all, we need algorithms that have enough *concurrency* to take advantage of large numbers of processors. Secondly, this concurrency must be *conveyed* in some language that will eventually be *translated* into commands that can direct the operations and cooperations of processors. Thirdly, the cooperation between processors may cause problems: when one processor needs to use another processor's result — it needs to find out whether that result will be ready to be used and where it can be found. The more processors a system has, the more *communications* between processors must take place. The danger is that the processors can spend more time telling each other how far they have gotten with their tasks than solving them. This processor bureaucracy must be brought under control. Another problem is the so-called *hot-spot* phenomenon where requests of services are concentrated at a few processors while others are waiting to be served. Like any large organization, concentration of workload must be smoothed out in order to achieve higher overall performance. Finally, there is a cost associated with communicating information between processors. The more local the communication, the less

costly. The benefit attained by taking advantage of *locality* must then be weighed against the cost and the effort spent in arranging for local communications and cooperations. Such arrangements may not always be made in a cost-effective manner. However, when a problem is amenable to such arrangement, the performance gain can be significant.

In this paper we describe how Crystal [5,6] — a language based on familiar mathematical notation and lambda calculus (see, for example, [19]) — addresses each of the above issues. What is new about Crystal (or how is it different from existing functional languages) lies in its model of parallel computation and a theory of parallel program optimization[7].

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

Regarding the suitable level of specification for concurrency, we don't encourage programmers to describe the microscopic behavior of each individual processor. Rather, you describe what is to be done by the processors collectively. This *lack* of specification in the source program about each processor's assignment provides the machine with the flexibility in interpreting how the collection of processors will execute the programmer's command.

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

Crystal's view of computation is of a *global* data space with non-uniform *communication metrics*. Each data subspace, called a *data field*, has its own shape and topology described as graphs. Code optimizations are specified as *morphisms* between data fields and carried out as source to source transformations. If so desired, a programmer can optimize the target code explicitly using data field morphisms. Again, this process can be carried out at the source level with the benefit of the mathematical apparatus. Implicitly by the compiler, or explicitly by a programmer, all optimizations are performed at the source level. Since the "target" Crystal program embodies information of the exact microscopic behavior of each individual processor, its translation to any of the many languages on parallel machines becomes a straightforward process.

The richness of the language in expressing shapes and topologies of data fields and the capability of abstracting away from implementation details imply the portability of source programs to different parallel architectures. Whether the target machine has shared-memory or distributed memory, is fine-grained or coarse grained, is small scale or massively parallel, the difference in

architectures simply induce different communication metrics for optimization.

The heart of Crystal's compilation process lies in morphisms for keeping communication overhead low (space-time realization, partition morphism), avoiding hot spot (fan reduction), ensuring balanced load (distributing morphism), and attempting locality whenever possible (contraction morphism). Crystal's theory of data field morphism says that all information for an optimization is embodied by a data field morphism, which is a pair of functions. These functions are in "closed form" for compile-time optimizations, and dependent on run-time data for run-time optimizations. Once a morphism is given, Crystal's metalanguage processor is responsible for either generating an optimized program or augment the original program with additional code that creates an environment for executing the morphism at run-time.

For a large program, the complexity of optimizations as well as programming must be managed. In Crystal, both can be done systematically by composing a large system in a hierarchical manner. Each Crystal sub-program can be separately designed and tested, and optimized by morphisms on the data field upon which it operates. When sub-programs are put together to create a larger one, optimizations for gluing their data fields together take place. We believe that this *structured optimization* approach may well be a more cost-effective one than the global optimization approach, especially when it comes to producing efficient target code for large, realistic applications.

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

2 Crystal System Overview

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

2.1 Optimization Library

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

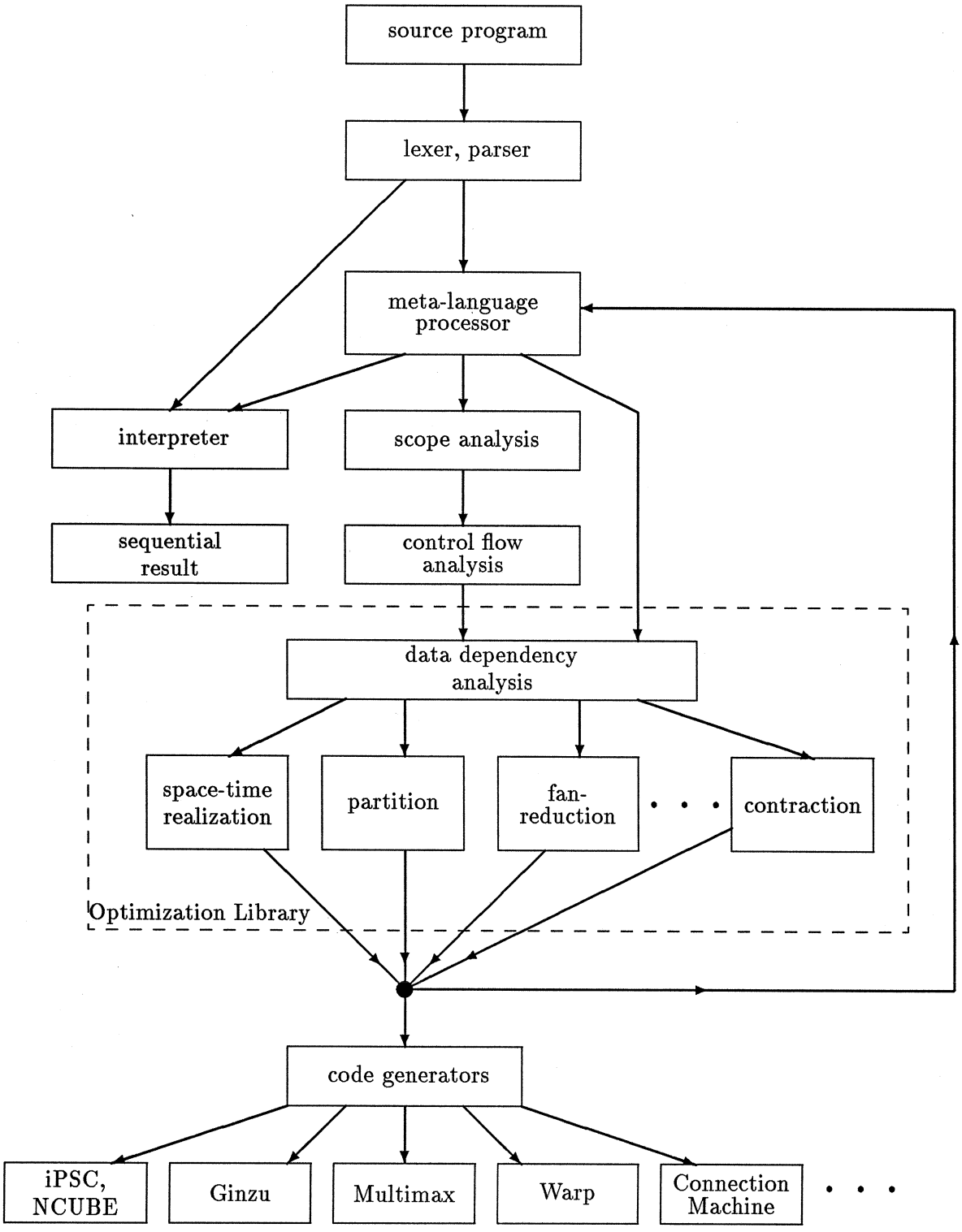


Figure 1: Organization of the Crystal Compilers

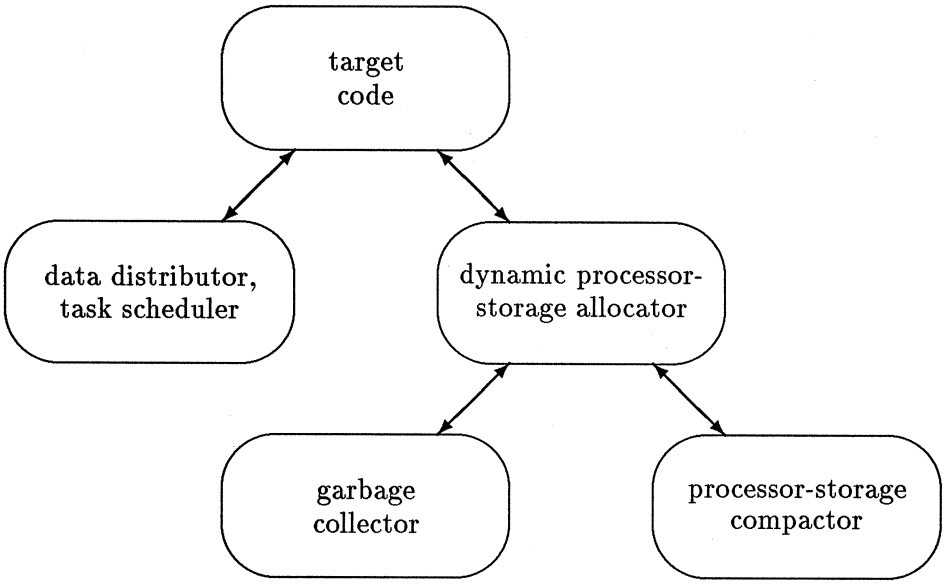


Figure 2: Run-time Environment of Crystal

2.2 Metalanguage Processor

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

2.3 Run-time Environment

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

2.4 Code Generators

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

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

3 The Crystal Language and Its Metalanguage

Crystal is a language for defining index domains, domain morphisms, data fields, and computation fields. It is basically the mathematical notation that was used for defining the various mathematical objects enriched with λ-abstraction, application, recursion, and local environment. 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.

In order to express high-level program transformations, a metalanguage is defined for Crystal, consisting of construtors, selectors and operators.

3.1 The Language

Data Types

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

The Conditional

The *conditional expression* has the following form

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

where the B_i's are boolean expressions and E_i's are any expressions, and 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 ([13]) 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 satisfies the equation

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

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

The *scan* ($\backslash\backslash$) is defined to be

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

and returns a data field of the same shape but 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 which augments the environment in which the definition is given. The definitions in a local environment may also have their own local environments.

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

3.2 A Metalanguage

The metalanguage will be used to formalize transformations of programs [7] 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 Crystal programs as objects [7].

Constructors, Selectors, and Predicates

For each construct of the object language, there exists a metalanguage constructor that takes the components and produces the construct.

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.

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

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

are the constructors and selectors for making equations and λ-abstraction. Other constructors include the eta-abstraction, application, and the conditional expression constructors.

Operators

Metalanguage operators manipulate the Crystal programs. The standard program transformations known as "fold" and "unfold" [4,8] take an equation (a function definition) and replaces a body by its function call, or the inverse. These can be defined in terms of more primitive operators "expand" which replaces the function name by its defining equation, and "reduce" which does β-reduction on a redex.

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

Other operators include substitution of a sub-term with an expression that is equal in the theory ($\mathsf{subst}(\kappa, \tau_1, \tau_2)$), and various simplifications for arithmetic expressions or function compositions.

The reshape operator, denoted $\mathsf{reshape}(\mathsf{def}(\ulcorner a \urcorner), \ulcorner g \urcorner, \ulcorner b \urcorner)$, is a composite function in the metalanguage. It takes an equation defining the function a and returns an equation defining the function b such that $a = b \circ g$ where g is an index domain morphism. The fan-in/fan-out reductions can also be expressed as metalanguage functions: $\mathsf{fan\text{-}out\text{-}red}(\mathsf{def}(\ulcorner c \urcorner), \ulcorner X_L \urcorner, \ulcorner a \urcorner)$ and $\mathsf{fan\text{-}in\text{-}red}(\mathsf{def}(\ulcorner c \urcorner), \ulcorner Z_L \urcorner, \ulcorner a \urcorner)$. The use of these will be illustrated below. A more complete presentation of the metalanguage is in [7].

4 A Model of Parallel Programs

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

The central notion in our model of parallel programs is that of a *data fields*, representing the evolution of data values distributed over a space and time of processors, known as *index domains*. 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 *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 different nodes of the domain. This allows us to model any kind of machine architecture, from single processors to shared memory machines, and distributed memory machines.

The communication metric provides us with a measure of a program's efficiency on a particular architecture and guides the strategies for parallel-program optimization.

4.1 Graphs and Communication Metrics

This section introduces notations for graphs and defines a communication metric over pairs of nodes. Graphs embody the topology of the computation. Morphisms between graphs allows us to reshape one computation into another that looks different yet is related to the original by the morphism.

Graphs and Graph Morphisms

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

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

Definition An *elementary surjection* is a mapping from one graph to another that collapses one arc with its source and target into one node while preserving paths. A *graph surjection* is a (possibly empty) sequence of elementary surjections. [11].

Definition A *graph morphism* is a composition of some graph injections and surjections.

In Crystal, graph morphisms are expressed as functions.

Definition For any graph morphism $g : G \to H$, a *left inverse*, if it exists, is a morphism $h : H \to G$ such that $h \circ g = 1_G$. Furthermore, if g is 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.*

Communication Metric

For this section, let G be a graph and let R be either the real or the natural numbers extended with infinity. We define the notion of a communication metric that measures the communication cost between nodes, and then show how a communication metric may be induced by communication cost for each arc.

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

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

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

Note, that a communication metric is weaker than the usual topological notion of a metric[9]. The usual metric is symmetric ($\mu(x, y) = \mu(y, x)$), and also satisfies the converse of condition 1 ($\mu(x, y) = 0$ implies $x = y$). A communication metric is meant to capture the communication cost along the paths of a graph. Therefore, the communication cost can be very different in opposite

directions. It may also be possible that two logical processors are mapped onto a single physical processor, resulting in the communication cost being zero.

In Crystal it will often be convenient to determine a communication metric by specifying the communication cost for each arc. We show that there is a canonical way to do so.

Definition A *local metric* on G is a function $\nu = \nu(G) : \rho(G) \rightarrow R$ that assigns to each arc a communication cost.

Given a local metric, it is straight forward to extend it inductively to a path so that for any path p, $\nu(p)$ is the sum of the local metric of each arc in the path.

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

4.2 Index Domains and Domain Morphisms

The graphs and communication metrics are used to define index domains and index domain morphisms.

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

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

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

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

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

For the following definitions, let D and E be index domains.

Definition A *domain injection* is a graph injection on the domains considered as graphs. A *domain surjection* is a graph surjection on domains considered as graphs. An *index domain morphism*, written $g : D \rightarrow E$, is some combination of domain injections and surjections.

Index domains with index domain morphisms form a category, since an identity morphism exists for each index domain and the composition of morphisms is associative.

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

A special class of index domains will be used to model discrete time.

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.

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

Note that $\tau(D)$ represents just one possible linearization of D.

Definition A *binary tree domain* is a domain with nodes connected as a binary tree. The predicates root and leaf test for root and leaves, and 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 some node.

Definition A *tree domain* over S with root r, denoted tree(S, r), is a binary tree domain with the leaves from S and the root r. If the cardinality of S is odd, we allow one non-leaf node not to have both children.

Definition A tree domain is *balanced*, denoted tree$_B$, if the leaves are all at the same distance from the root, is *left-associative tree domain*, denoted tree$_L$, if all the right children are leaves, and is *right-associative*, denoted tree$_R$, if all the left children are leaves.

Definition Let $g : D \to E$ be a domain morphism. Then the *induced* communication metric on the domain D by g is defined by

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

Domain Constructions

Given domains, new domains can be constructed. The usual cartesian product and disjoint union are defined, and a special construction called time product is introduced.

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

Example

$$\lambda x : \{\, 0 \,..\, 100 \,\} \& even(x).5x$$

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

Definition Let D_1 and D_2 be domains. The *cartesian product* of D_1 and D_2, denoted $D_1 \times D_2$, is defined as follows:

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

where the local metric is

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

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.

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

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

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

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

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

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

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

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

where

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

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

Figure 3: S T $S \times T$ $S * T$

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.

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

Example Let S be a three element domain and T be a three element domain. Their cartesian product and time product are depicted in Figure 3.

Generalized Products and Coproducts

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, and let P be a restriction on I.

The *product* of $\{ D(i) \mid i : I{\downarrow}P \}$ is denoted $\prod i : I{\downarrow}P.D(i)$, and the *coproduct* of $\{ D(i) \mid i : I{\downarrow}P \}$ is denoted $\sum i : I{\downarrow}P.D(i)$.

Let $D = \sum i : \{ 0 \,..\, n \}.\sum j : \{ 0 \,..\, i \}.\{ (i, j) \}$ denote a triangular domain. Then any function defined over it can be thought of as implementing a parallel version of a nested loop, where the inner j is bound by the outer i loop variable.

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

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

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

4.3 Data Fields and Data Field Morphisms

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

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

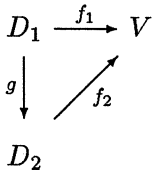
A data field $a : D \to V$ can be expressed as a function over the index domain. For example, let $D = \{ 0 .. 20 \}$, then

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

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

Index domain morphisms are used to define morphisms of data fields.

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



commute.

Parallel-program optimization consists of reshaping the data field so that it will be most efficient for a given space of processors. Reshaping of data fields consists of finding an isomorphism between two index domains. We describe the important ones here.

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 called an *affine morphism.*

Let $D = \{ 0 .. n - 1 \}$, $E = \{ 0 .. 2n - 1 \}$, and $T = \{ 0 .. 3n - 2 \}$ be domains. An example of an affine morphism is the spacetime realization

$$g = \lambda(i, j, k) : D^3.(i + k, j + k, i + j + k) : (E \times E) * T$$

and the communication metric on $E \times E$ is induced by a local metric that assigns the unit cost to the arc joining each node to ones lying in the direction of the vector $(0, 1), (1, 0), (-1, -1)$ from it.

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 $g : D \to \sum i : I.D(i)$.

Contraction

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

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

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

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

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

In general, the codomain of a contraction may be the coproduct of many copies of E's.

Note, that once the linear morphism or contraction g and g^{-1} have been given, there is a purely formal way to derive the layers d_1 and d_2 from a. A worked example of each will be given below.

Spacetime Realization

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

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

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

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

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

1. Minimize time. Pick $\langle s, t\rangle$ 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 $\langle s, t\rangle(C)$ inside the minimum bounding volume $S * T$.

Refinement and Abstraction

Given an injection $g : D \to E$, we think of E as containing more structure than D. Conversely, if $h : E \to D$ is a surjection, then D can be thought of as an abstraction of some details in E. We make this notion precise by requiring that the $g \circ h = 1_D$.

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 $g \circ h \sqsubseteq 1_E$.

For 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.

When $h = g^{-1}$ in a refinement morphism, we shall call it a *reshape morphism*, and not explicitly indicate the inverse.

We overload and say that E is the *refinement* of D along g, and D is the *abstraction* of E along h.

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

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

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

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

Program transformations corresponding to fan-in and fan-out refinement are described next.

Fan-in Reduction Consider the program

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

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

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

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

where

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

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

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 fan-out-red(def($\ulcorner c \urcorner$), $\ulcorner X_L \urcorner$, $\ulcorner a \urcorner$).

Currying

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

For index domain D and a set of values V, let $[D \rightarrow V]$ to be the set of all data fields over D taking values in V.

Proposition 4.1 *For any index domains D and E and a set of values V, the function*

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

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 to $\phi(f)$ is known as *currying.*

Currying corresponds to the interchanging of loops, in sequential program optimization.

The task of coming up with suitable refinement morphisms is non-trivial, but for large class of problems, the regularity of the computation makes automatic generation of certain morphisms possible.

5 Examples of Compile-time Optimizations

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

Program 1 (Matrix Multiplication)

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

Program 2 (LU-Decomposition)

$$\begin{aligned} \textit{lu-decomposition}(A0, n) &= [L, U] \\ &\text{where } \{ \\ a(i,j,k) : D_a &= \left\{ \begin{array}{l} k = 0 \rightarrow A0[i-1, j-1], \\ 0 < k \rightarrow a(i,j,k-1) - L(i,k) * U(k,j) \end{array} \right\}, \\ L(i,k) : D_l &= \left\{ \begin{array}{l} i < k \rightarrow 0, \\ i = k \rightarrow 1, \\ k < i \rightarrow a(i,k,k-1)/U(k,k) \end{array} \right\}, \\ U(k,j) : D_u &= \left\{ \begin{array}{l} j < k \rightarrow 0, \\ k \leq j \rightarrow a(k,j,k-1) \end{array} \right\} \\ &\quad ! \textit{ define a 3-dimensional domain} \\ D &= \{1 .. n\} \times \{1 .. n\} \times \{1 .. n\}, \\ &\quad ! \textit{ projection of } D \textit{ along the 1'st dimension} \\ D_l &= \text{proj}(1)(D), \\ &\quad ! \textit{ projection of } D \textit{ along the 0'th dimension, then transpose the domain} \\ D_u &= \text{trans}(\text{proj}(0)(D)), \\ &\quad ! \textit{ coproduct of two domains} \\ D_a &= D + (\text{proj}(2)(D) \times \{0\}), \\ &\} \end{aligned}$$

From this level of description, target C code plus communication commands are compiled. We illustrate in the following the compiler optimization steps that transform the matrix multiplication program into a target Crystal program that yields the blocked partition program described in Section 6.

5.1 Fan-in reduction

In Program 1, hot spots occurs at each index $(i,j) \in D$ since n terms are summed together. By performing fan-in reduction fan-in-red(def($\ulcorner C \urcorner$), $\ulcorner Z_L \urcorner$, $\ulcorner \widehat{C} \urcorner$) the new program becomes:

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

Expanding the definitions of the Z_L, we obtain

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

Expanding the definitions of particular implementations of leaf, left and right of $\mathrm{tree}_L(N_0, r)$ (where a left-associative tree is collapsed into an array N_0), and the definition of H, we obtain

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

5.2 Partition Morphism

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

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

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:

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

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

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

5.3 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 $\hat{c}$:

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

5.4 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

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

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

$$\hat{c}_A = \lambda u1 : U_{b1}.\lambda u0 : U_{b0}.\lambda(v0,v1) : V_{b0} \times V_{b1}).\lambda k : N_0.$$

$$\left\{\begin{array}{l} k = 0 \rightarrow 0, \\ \neg(k = 0) \rightarrow \hat{c}(u0, u1)(v0, v1)(k-1) \\ \qquad\qquad +a(u1)(u0, v0, k) \times B(k, (u1 \times b1 + u1)) \end{array}\right\}$$

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

Fan-out reduction for matrix elements of B can be carried out similarly. The resulting Crystal program is then translated to parallel C code.

The second example (coarse-grained systolic version) described in Section 6 requires modification of the above step and a few additional optimization steps, which can be summarized as follows:

1. Instead of fan-out reduction by balanced tree tree_B in the above step, use left-associative tree tree_L.
2. Partition morphisms applied to definitions of a and b.
3. Currying
4. Affine morphism for achieving a better utilization of processors.

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

6 Performance of Compiler-Generated Code

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

6.1 Measuring performances

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

- *Computation statements:* Any statements which are associated with local computations.
- *Communication statements:* Statements which involve inter-processor communications, for instance, send and receive statements, statements for preparing message buffers, statements for unpacking incoming messages, and etc.

- *Initialization statements:* Statements for initializing variables, array allocations, opening communication channels, etc. that are executed only once in each invocation of the node program.

The cpu time spent on the execution of a node program is collected in two parts: computation time and communication time. The cpu time spent on the initialization statements are negligible, and can be ignored. For each processor i, we define:

- $t(i)$ (*Elapsed time*): time spent on the entire execution.
- $t_{compt}(i)$ (*Computation time*): time spent on computation statements.
- $t_{comm}(i)$ (*Communication time*): time spent on communication statements. It also includes the idle time due to waiting for other processors' messages.

$$t(i) = t_{compt}(i) + t_{comm}(i).$$

Performance of a program on the iPSC (or the NCUBE) is measured by its over all elapsed time, computation time, and communication time. Speedup of parallel execution over sequential execution is also computed for each case. These quantities are defined below:

- **Number-of-Processors:** N.
- **Sequential-Elapsed-Time:** $T_s = t_{compt}(1)$, where $N = 1$. (The program is executed on one single processor, $t_{comm}(1) = 0$.)
- **Parallel-Elapsed-Time:** $T_e = \max_{1 \le i \le N} t(i)$.
- **Average-Computation-Time:** $T_p = (\sum_{i=1}^{N} t_{compt}(i))/N$.
- **Average-Communication-Time:** $T_m = (\sum_{i=1}^{N} t_{comm}(i))/N$.
- **Speedup:** $r = T_s/T_e$.
- **Efficiency:** $e = r/N$.

6.2 Performance Analysis

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

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

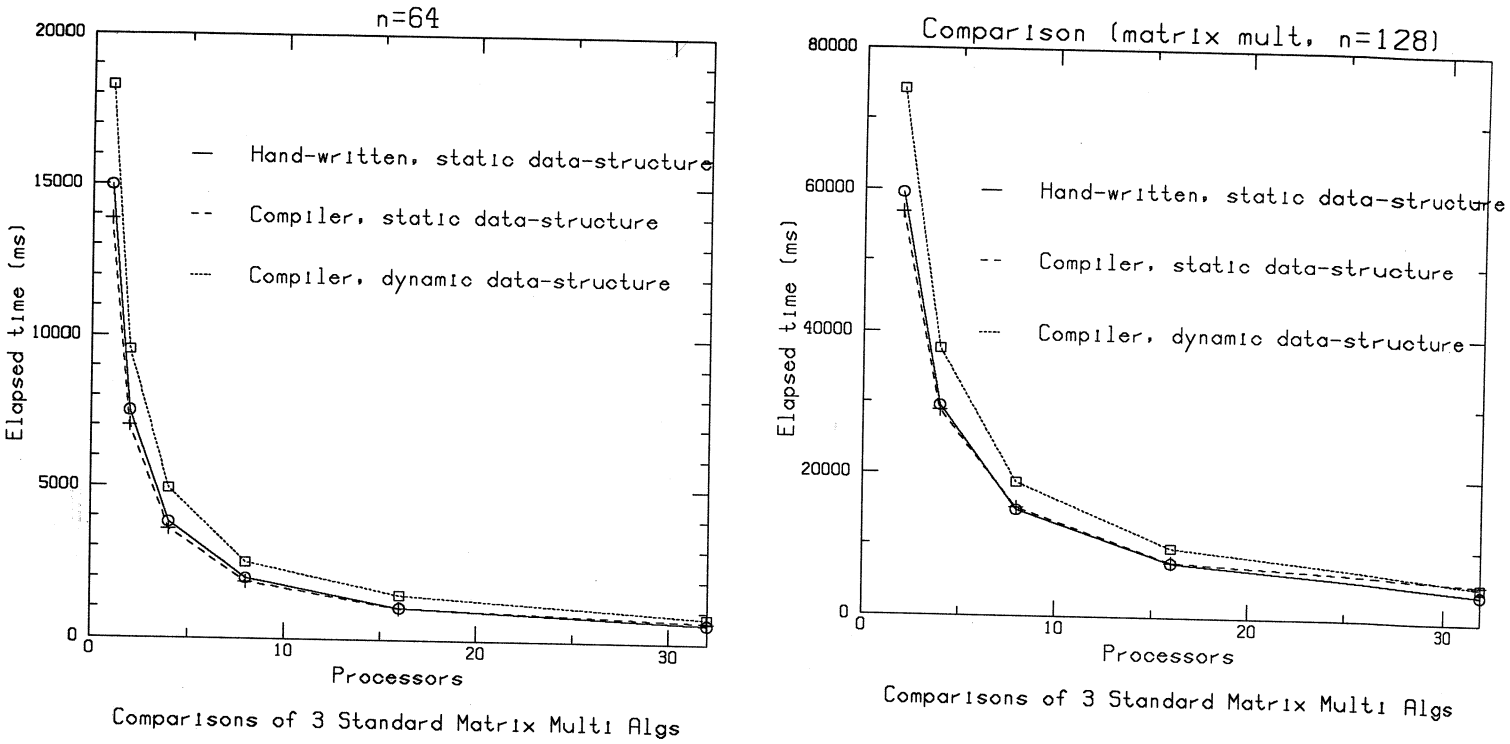


Figure 4: Performances of three block MM programs on iPSC. Two are generated by compiler; one is manually written.

2. **Coarse-grained systolic matrix multiplication program**: Here, elements of matrices A and B are pipelined through processors. The program is organized in an iteration loop. In each iteration, every processor does three things: receiving some data from its neighbors, passing some data to its neighbors, and accumulating some partial results locally. At the end, every processor will have a piece of the product matrix C.

3. **Coarse-grained systolic LU decomposition program**: It decomposes a square matrix A into its L, U factors. This program is structurally the same as the systolic matrix multiplication program.

Compiler Generated Program vs. Hand-Written Program

Figure 4 shows performances of three matrix multiplication programs, which use the same block partitioned algorithm. Two of them are generated by the Crystal compiler. The other is manually written. One of the compiler-generated programs uses static memory allocation, i.e. array sizes are runtime constants. The other uses dynamic memory allocation, i.e. arrays are allocated by *malloc()* at runtime. The hand-written program uses static memory allocation.

In Figure 4 we see that the compiler-generated programs perform almost as good as the hand-

written one. The overheads introduced by the compiler are negligible.

Systolic Communication vs. Broadcasting

The two versions of the matrix multiplication program employ different types of communications. For the systolic matrix multiplication program, communications only exist between adjacent processors. This communication locality is very suitable for the iPSC and the NCUBE, for on both machines, the communication cost is proportional to the distance between the source and the destination processors. Another good feature of communication locality is that the communication cost does not scale up with the size of the machine. If the problem size is fixed, increasing the number of processors in a multiprocessor will reduce the sub-problem size on every processor. This reduces the amount of messages that every processor has to communicate with. Therefore, increasing the number of processors will reduce communication time for each processor.

For the block-partitioned matrix multiplication program, broadcasting among many processors (a row or a column) is needed. A broadcasting is done by dynamically creating a communication tree and exchanging data at each level of the tree. The time a broadcasting takes is $\log m$, where m is the number of processors involved. (Cf. [14] for detailed discussions about optimal broadcasting on hypercubes.) When the number of processors is increased, the communication time may not decrease, since although the amount of messages on each processor will be reduced, the cost of broadcasting will increase.

In Figure 5, we see that for the systolic program, both T_p and T_m decrease when N increases. For the block-partitioned program, T_m stays roughly at the same level, no matter what N is. Therefore, for the block-partitioned program, when N increases, more percentage of cpu time will be spent on communication. The speedup of parallel-elapsed-time over sequential-elapsed-time will not grow linearly with the increase of N, as shown in Figure 6.

Effects of Changing Granularity

When a program is organized in an iteration loop (like the systolic matrix multiplication program or the systolic LU decomposition program), a parameter is provided by the compiler to control the size of an iteration step between two inter-processor communications.

The first effect of changing granularity is on communication time. To illustrate this, we write the formula for computing t_{comm} as follows:

$$t_{comm} = k \times (\text{startup-time} + \text{msg-size} \times \text{transmitting-time})$$

where k is the total number of iteration steps.

A larger iteration step means less iterations will be needed and messages will be grouped in larger packages. This will reduce the overhead of communication startup time, therefore reduce t_{comm}. Thus, a coarser grain implies less communication overhead.

However, changing granularity can also effect the delay time. In the first phase of an iteration, if the computation on a processor depends on results from another processor, then the first processor has to wait for the second one to finish its first iteration. The waiting time is called delay time. The

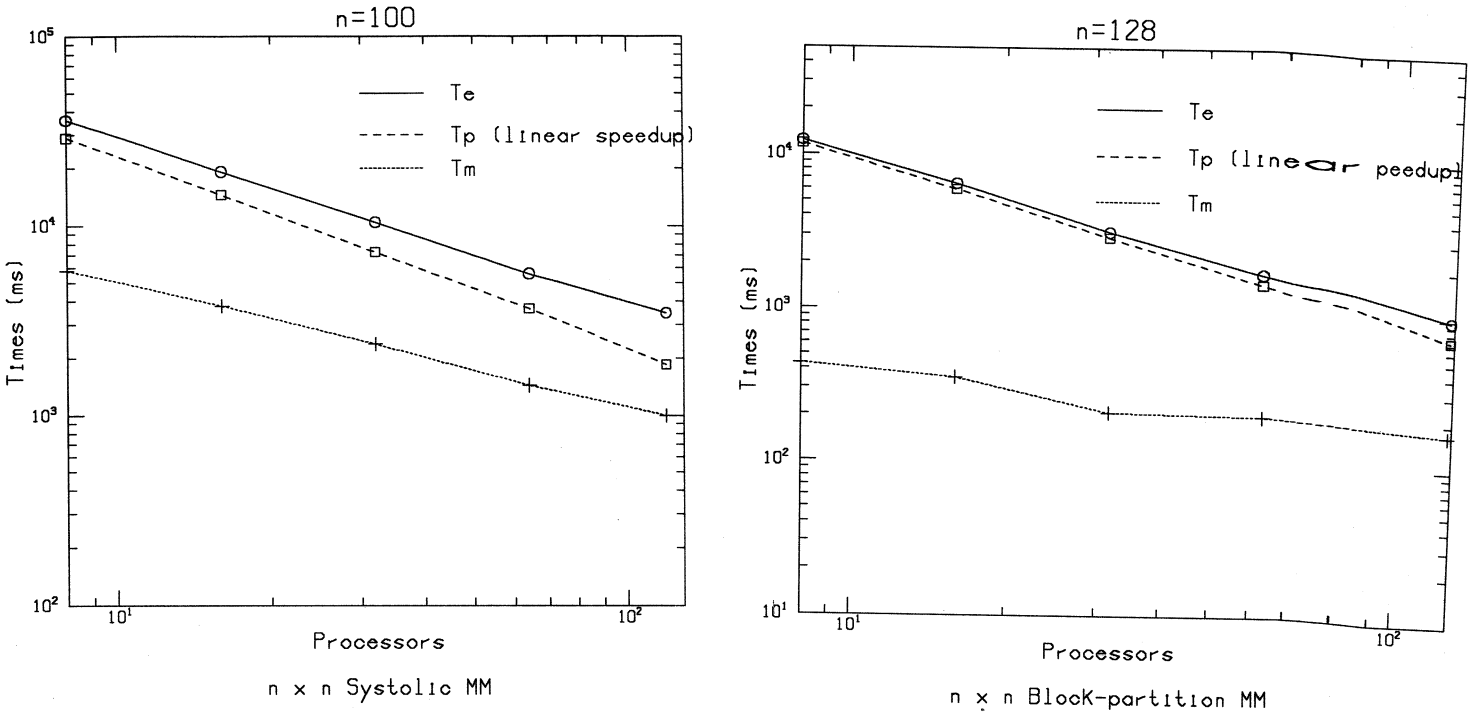


Figure 5: Performances of two different types of matrix multiplication program: systolic and block-partitioned, on NCUBE.

delay time, apparently depends on the amount of computation in each iteration. A larger iteration step, therefore, means larger delay time.

Adjusting granularity is actually a trade-off between the communication overhead and the delay time. Where the best point lies, depends on both machine's and program's characteristics.

The following table shows one example.

Step size	50	25	17	13	10	7	5	3	2	1
T_e	21503	16217	14536	13720	13123	12626	12295	12208	12392	13423

Table 1. Effects of granularity on elapsed-time. Systolic matrix multiplication program on NCUBE. Matrix-size = 50×50, $N = 4$. The best step size for this program is 3.

iPSC vs. NCUBE

All of the three programs have been run on an iPSC and an NCUBE. The following is a comparison between the two machines.

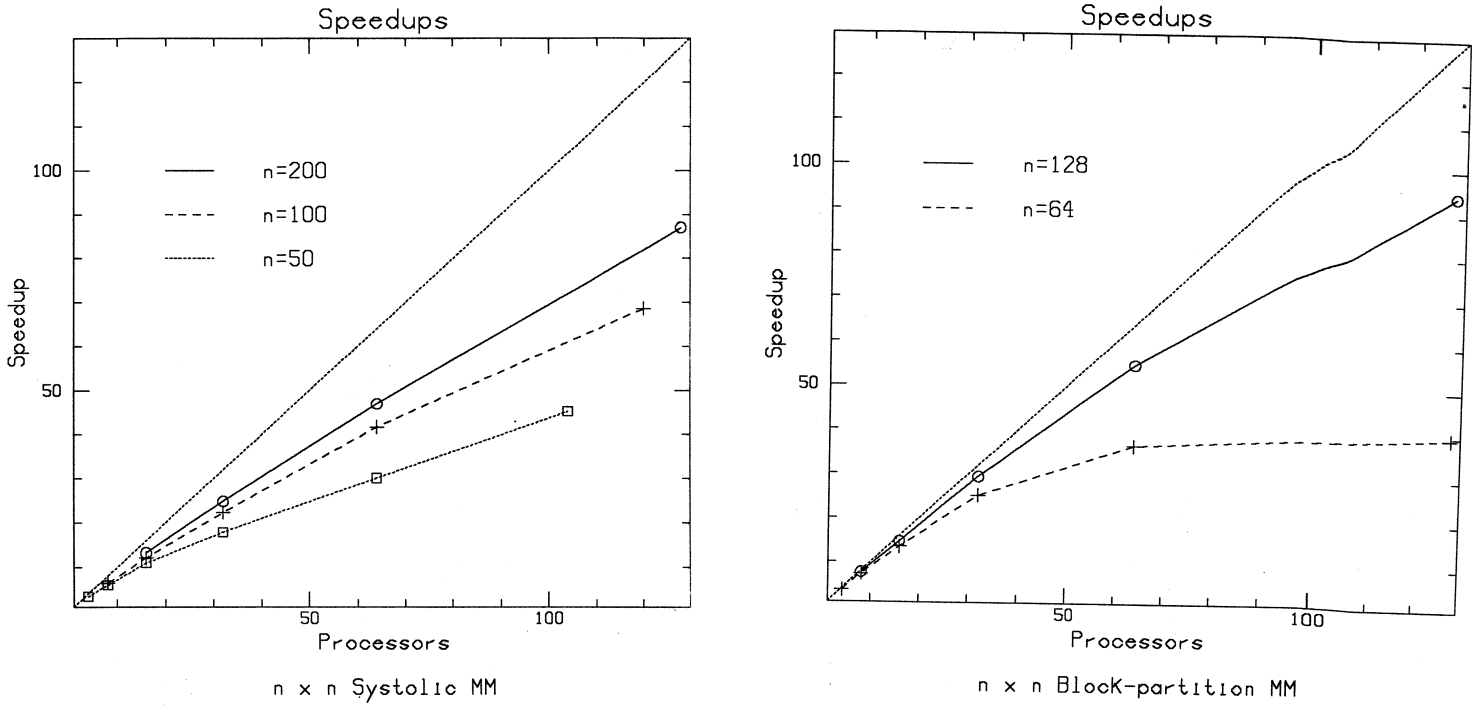


Figure 6: Speedups of the systolic, and the block-partitioned matrix multiplication programs on NCUBE.

iPSC						
program		N	T_p	T_c	T_e	r
Systolic MM	100x100	32	10698	3330	14042	22.7
Block MM	100x100	32	2103	340	2472	22.6
Systolic LUD	100x100	32	11777	3455	16695	22.6

NCUBE						
program		N	T_p	T_c	T_e	r
Systolic MM	100x100	32	7254	2382	10420	22.3
Block MM	100x100	32	1386	136	1620	27.4
Systolic LUD	100x100	32	8138	2670	11860	22.0

We can see in the tables that for the timing entries (i.e. T_p, T_c, and T_e), numbers in the second table are all less than their corresponding ones in the first table. The reason for this is because

the NCUBE' processors are more powerful than the iPSC's processors. However, there is no big differences between the corresponding speedup numbers in the two tables. This means that the computation/communication cost ratios on the two machines are very close.

The performance of the target code on the largest hypercube used in our experiments is summarized as follows:

NCUBE						
program		N	T_p	T_c	T_e	r
Systolic MM	200x200	128	14317	5075	21060	87.0
Block MM	128x128	128	721	171	980	94.2
Systolic LUD	200x200	128	16240	5721	24220	85.8

7 Concluding Remark

The parallelizing compiler approach and Crystal's approach to parallel processing attack the same problem from two opposite perspectives: one attempts to increase the degree of parallelism while the other reduces it. Nevertheless, many of the techniques used in parallelizing Fortran compilers [1,10,16,15,3,17,18] can be understood as morphisms. For example, *strip mining*[16], a technique for memory management, transforms a single sequential loop into a double nested one so that the outer loop of the resulting code can be executed in parallel. The partition morphism of Crystal, in contrast, transforms a one-dimensional data field, where all elements can be executed in parallel, into a two-dimensional data field. The computation along one of the dimensions can then be serialized in order to reduce communication overhead. Crystal's fanout reduction and contraction morphisms are particularly useful for distributed-memory machines where the cost of accessing data is not uniform.

Much effort has been made in the realm of functional languages [2] and data flow processing (see summary in [20]) for discovering latent parallelism in programs. Due to the freedom from side-effects, functional calls that are not interdependent can be executed in parallel. But the focus of data flow processing has been on parallel tasks with different threads of control. The distribution (copying) of data to parallel tasks therefore becomes a source of inefficiency. Remedies such as explicit specifications of data locations by annotations are used in [12]. Though Crystal is also a functional language, it tackles the data distribution problem at the most fundamental level, namely, in the model. The notions of data fields and communication metrics are essential for the compiler optimization that generates efficient code.

Because we are free from the constraints of an existing language, we can make Crystal and, most importantly, its model to be algebraic. Consequently, optimization techniques are captured as morphisms, which can be expressed as Crystal functions and carried out by the metalanguage processor in a uniform and systematic fashion. We also believe that the theoretical foundation helps us in managing the complexity of implementing Crystal by factoring out the part of the system that contains heuristics (the optimization library) and the part of system which is entirely algebraic. Currently, we are planning the programming of large, realistic applications in Crystal.

We hope to demonstrate that elegance, practicality, and high performance are not competing goals but synergistic ones.

Acknowledgment We would like to thank Eileen Connolly for her editorial effort and the generous support by the Office of Naval Research under Contract No. N00014-86-K-0564.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

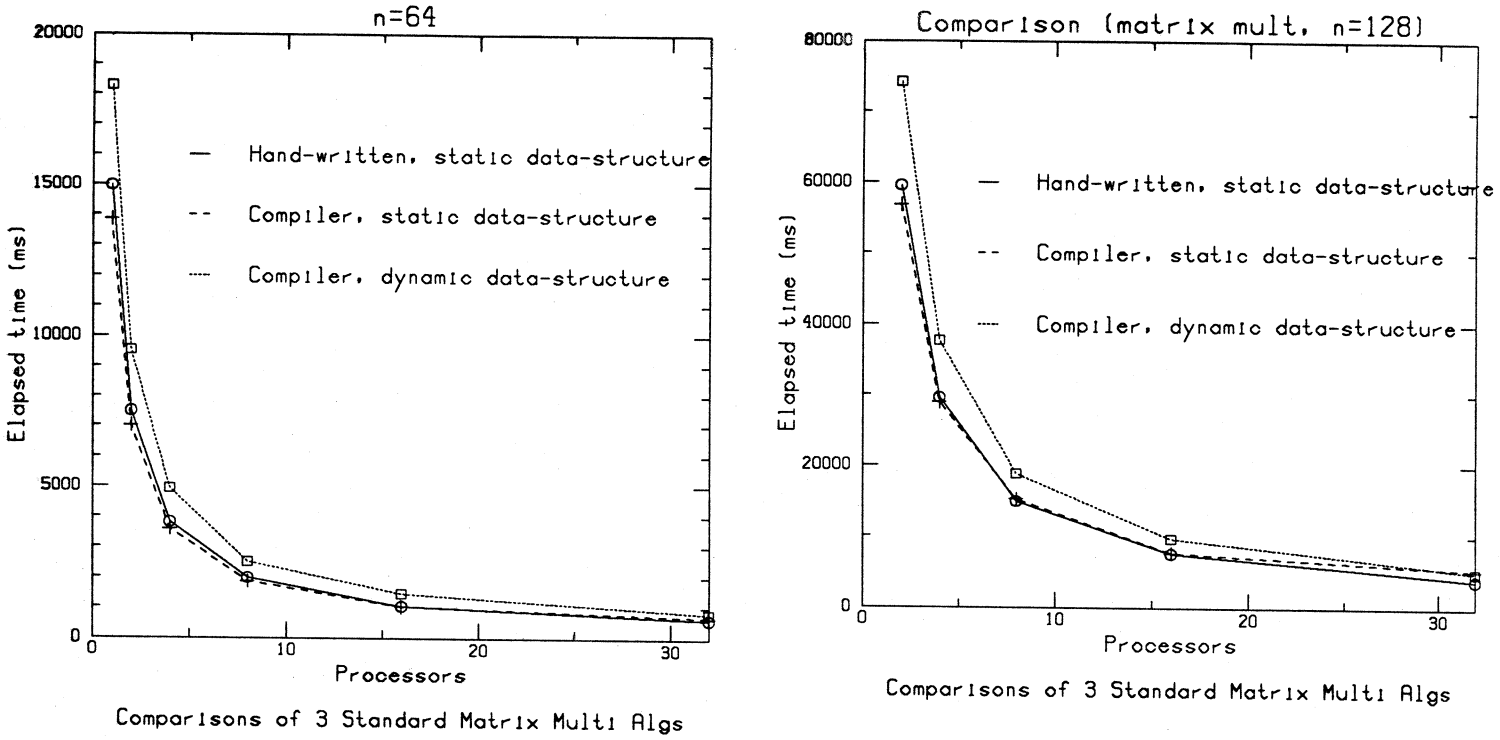


Figure 4: Performances of three block MM programs on iPSC. Two are generated by compiler; one is manually written.

2. **Coarse-grained systolic matrix multiplication program**: Here, elements of matrices A and B are pipelined through processors. The program is organized in an iteration loop. In each iteration, every processor does three things: receiving some data from its neighbors, passing some data to its neighbors, and accumulating some partial results locally. At the end, every processor will have a piece of the product matrix C.

3. **Coarse-grained systolic LU decomposition program**: It decomposes a square matrix A into its L, U factors. This program is structurally the same as the systolic matrix multiplication program.

Compiler Generated Program vs. Hand-Written Program

Figure 4 shows performances of three matrix multiplication programs, which use the same block partitioned algorithm. Two of them are generated by the Crystal compiler. The other is manually written. One of the compiler-generated programs uses static memory allocation, i.e. array sizes are runtime constants. The other uses dynamic memory allocation, i.e. arrays are allocated by *malloc()* at runtime. The hand-written program uses static memory allocation.

In Figure 4 we see that the compiler-generated programs perform almost as good as the hand-

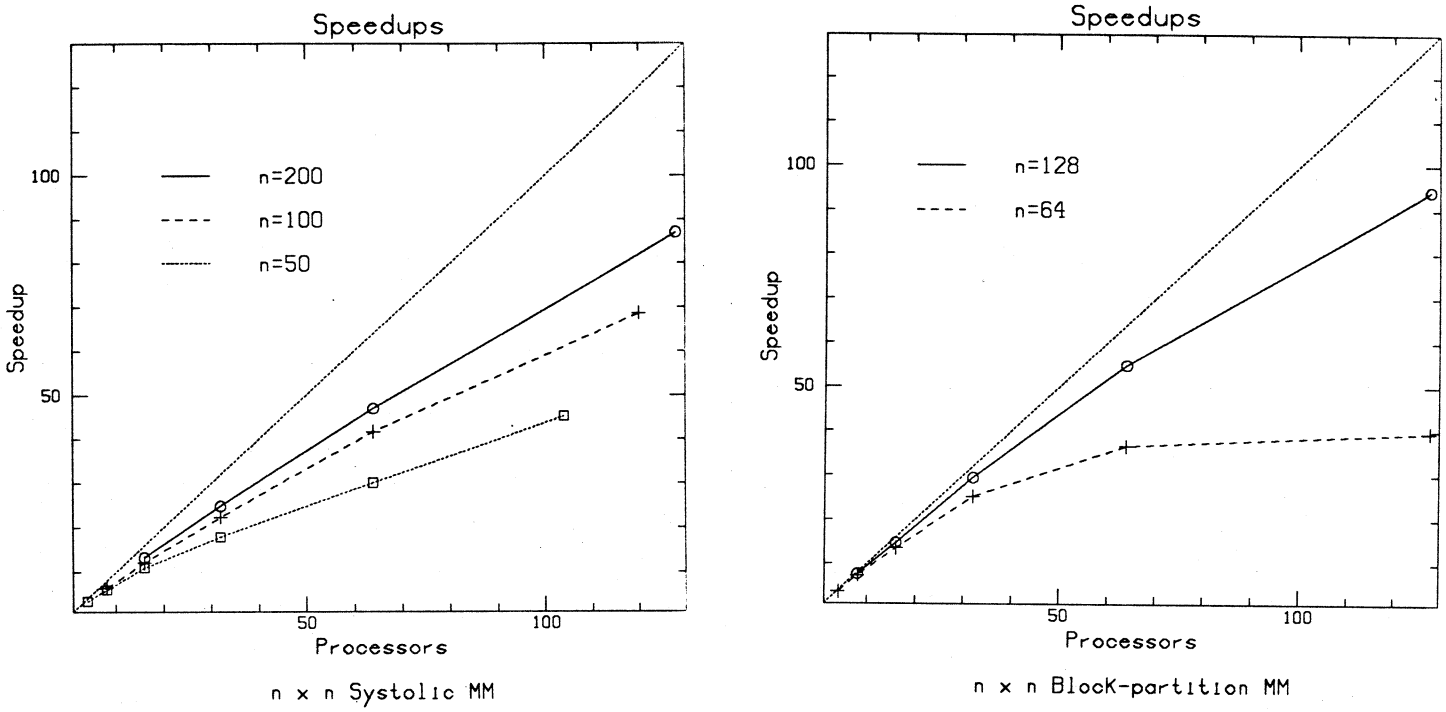


Figure 6: Speedups of the systolic, and the block-partitioned matrix multiplication programs on NCUBE.

iPSC						
program		N	T_p	T_c	T_e	r
Systolic MM	100x100	32	10698	3330	14042	22.7
Block MM	100x100	32	2103	340	2472	22.6
Systolic LUD	100x100	32	11777	3455	16695	22.6

NCUBE						
program		N	T_p	T_c	T_e	r
Systolic MM	100x100	32	7254	2382	10420	22.3
Block MM	100x100	32	1386	136	1620	27.4
Systolic LUD	100x100	32	8138	2670	11860	22.0

We can see in the tables that for the timing entries (i.e. T_p, T_c, and T_e), numbers in the second table are all less than their corresponding ones in the first table. The reason for this is because

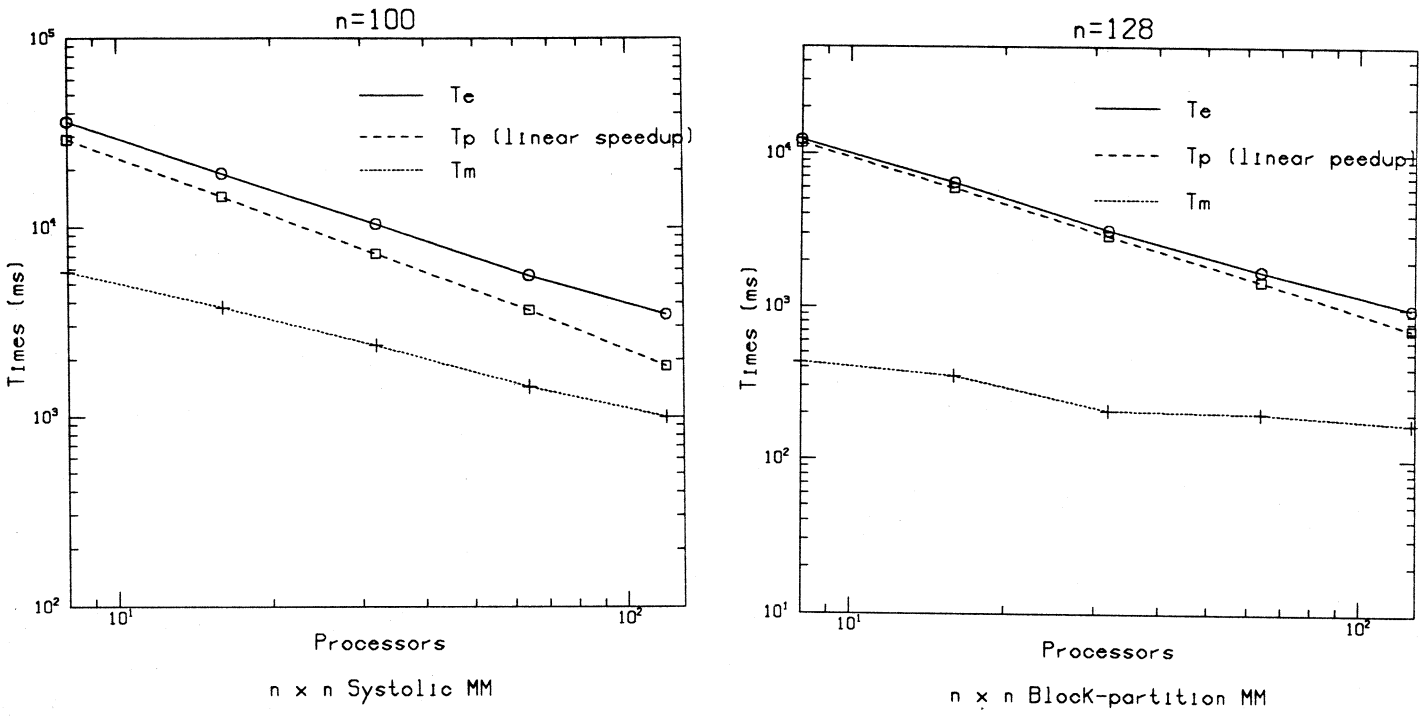


Figure 5: Performances of two different types of matrix multiplication program: systolic and block-partitioned, on NCUBE.

delay time, apparently depends on the amount of computation in each iteration. A larger iteration step, therefore, means larger delay time.

Adjusting granularity is actually a trade-off between the communication overhead and the delay time. Where the best point lies, depends on both machine's and program's characteristics.

The following table shows one example.

Step size	50	25	17	13	10	7	5	3	2	1
T_e	21503	16217	14536	13720	13123	12626	12295	12208	12392	13423

Table 1. Effects of granularity on elapsed-time. Systolic matrix multiplication program on NCUBE. Matrix-size = 50 × 50, $N = 4$. The best step size for this program is 3.

iPSC vs. NCUBE

All of the three programs have been run on an iPSC and an NCUBE. The following is a comparison between the two machines.