Compiling Crystal for Distributed-Memory Machines

Jingke Li

YALEU/DCS/RR-876

October 1991

Compiling Crystal for Distributed-Memory Machines

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Jingke Li

October 1991

Abstract

Compiling Crystal for Distributed-Memory Machines

Jingke Li

Yale University

1991

This dissertation presents a set of novel compilation techniques and the overall design of a compiler for distributed-memory machines. It demonstrates that efficient code can be automatically generated from machine-independent shared-memory-based programs for massively parallel distributed-memory machines.

Compiling programs for distributed-memory machines faces great challenges. This dissertation formulates precisely several key compilation problems, analyzes their complexity, and presents practical algorithms for solving them. For the problem of partitioning and distributing data to the processors, a technique called index domain alignment is developed. It aligns a group of arrays in a way that minimizes data movement caused by cross-references between these arrays. For the problem of generating interprocessor message-passing, a novel approach based on pattern matching is developed that enables the compiler to generate efficient collective communications of various forms. The dissertation also presents an algorithm for transforming a functional source program into an imperative intermediate program consisting of explicit constructs for representing parallelism and control information.

In addition to developing and studying these techniques, this dissertation presents the complete design of a compiler that transforms a program written in a high-level functional language, Crystal, into C programs augmented with communication primitives. A prototype of the Crystal compiler has been implemented for the Intel iPSC/2 and the nCUBE I hypercube multiprocessors. Experimental results for a variety of applications are included.

Acknowledgement

I would like to thank my advisor, Marina Chen, for several years of guidance, encouragement, and support. Without her leadership of the Crystal Project, this dissertation would not have been possible. I would also like to thank the other members of my committee. Ken Kennedy offered great encouragement and helpful advice to my work and did a timely and thorough reading of this dissertation despite his busy schedule. Young-il Choo and Ron Pinter both gave valuable advice which influenced both the content and form of this dissertation.

I am grateful to Joe Rodrigue, Cheng-Yee Lin, Janet Wu, Lakshmi Dasari, and Seema Hiranandani for their contributions to the Crystal hypercube compiler. I am also thankful to Michel Jacquemin, Lee-Chung Lu, Aloke Majumdar, Allan Yang, Chris Darken, Magne Haveraaen, Winnie Shu, and Min-You Wu for many helpful discussions. Chris Hatchell, Joe Rodrigue, and Sidd Puri kindly proofread this dissertation and offered many constructive comments and suggestions.

I would like to thank Bill Gropp and Martin Schultz for helping me improve my understanding of scientific computing problems. I would also like to thank Kay Crowley and Erik DeBenedictis for helping me with the Intel and nCUBE hypercube machines.

I am indebted to Michael Wolfe for reading and commenting on this dissertation, to Len Shapiro for supporting the final revision, and to my long-time friend Mark Young for answering my numerous questions concerning this dissertation as well as American culture in general.

I would like to thank all of my friends at Yale for making my life in New Haven an unforgettable experience. My deep gratitude goes to Xiaoyan for her persevering love and support throughout the years.

Finally, I sincerely thank my parents for giving me their wholehearted support for all these years. I dedicate this dissertation to them.

This research was supported in part by Office of Naval Research contracts N00014-86-K-0310, N00014-86-K-0564, N00014-89-J-1906, and N00014-90-J-1987, and by an National Science Foundation grant CCR-8908285.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The great advances of computer technology have made massively parallel, distributed-memory computers a reality. Commercially available machines now have up to thousands of processors per machine, and there seems no limit on how large these machines can ultimately be. Potentially, a massively parallel machine can out-perform a sequential supercomputer of comparable cost by several orders of magnitude. Despite these promises, however, massively parallel machines have not yet entered the mainstream of computation. The main obstacle is the difficulty of programming them—sequential programs cannot directly be executed on these machines and the currently available programming tools require the user to control explicitly algorithm-irrelevant, low-level issues in application programs. Ironically, this level of programming resembles writing assembly programs for a sequential machine.

This dissertation presents a solution to this mismatch of hardware and software. We take a compilation approach in which we allow the user to program an application in a high-level language, and the compiler automatically generates a target program which contains all the necessary details for execution.

1.1 Distributed-Memory Machines

A distributed-memory machine consists of a set of identical processors linked by an interconnecting network. Each processor has a private memory, which is accessible only by that processor. There is no global shared memory, hence data sharing and task coordination among processors are accomplished through message passing. Common network topologies that are used to link processors in a distributed-memory machine include hypercube, mesh, and butterfly. In each case, every processor has direct links to a bounded number of other processors and all communications between non-neighboring processors are routed through intermediate processors.

A distributed-memory machine is a *coarse-grained* machine if the processors are powerful and the private memories are large. It is a *fine-grained* machine if it consists of a massive number of small processors. Examples of coarse-grained machines include Intel's iPSC/2 and iPSC/860, nCUBE's nCUBE 2. Examples of fine-grain machines include Thinking Machine's CM-2, and the MasPar's MP-1.

Writing efficient programs for distributed-memory machines is a great challenge for a programmer. Many issues that do not arise in programming shared-memory machines (including sequential machines) must be addressed in programming distributed-memory machines:

- *Parallelism:* Parallelism in the application must be made explicit in the user program. Generally speaking, this involves breaking a computation into a collection of *parallel tasks*, which are assigned to different processors and executed in parallel.

- *Data Layout:* Since there is no global shared memory on a distributed-memory machine, large data structures in an application must be partitioned and distributed over the processors.

- *Communication and Coordination:* Distributed tasks which coordinate a joint computation often need to share data or to synchronize operations with each

other. To realize this, explicit message passing must be constructed and inserted into the user program.

The most naive approach to programming a distributed-memory machine is to program each sequential processor with a conventional language and insert explicit communication commands in the program for sending and receiving messages. The advantage of this approach is that the programming model matches well with the machine architecture, hence it requires few new compilation techniques to implement a language of this model. In fact, almost all of the commercial machines support languages of this model. For instance, C and Fortran with message-passing extensions are available on Intel's and nCUBE's hypercubes and Occam is supported on Transputer-based machines.

To the programmer, however, programming a distributed-memory machine in this fashion is comparable to programming a sequential computer at the assembly level. Many machine related details, such as partitioning data among processors and sending and receiving messages, have to be specified in the user program, which can be tedious and error-prone. Furthermore, the programs written this way are not portable across different machines, sometimes not even across different sizes of the same kind of machine.

The difficulty in directly programming a distributed-memory machine is largely due to the need to explicitly manage data distribution and communication. One solution to overcoming this difficulty is to build a *virtual shared memory* on top of a distributed-memory machine. As a result, the user programs in a virtual shared-memory environment. In many ways, this is easier than programming the machine directly. However, providing uniform access to an arbitrary memory will inevitably slow down the unit access time. The overhead of supporting global memory on a distributed-memory machine, especially when the underlying machine consists of a massive number of processors, may greatly reduce the benefit that this approach promises.

A second solution to the programming challenge is to develop *a smart compiler* to

do the hard work. This would enable the user to program in a high-level language and to express algorithms rather than machine-related details in the application program. Relying on a compiler for handling details has been the standard approach for programming sequential machines. Fortran, C, and Pascal are all machine independent, high-level languages. Building a smart compiler for distributed-memory machines is, nevertheless, all but a trivial task. Conventional compilation techniques provide little help for solving problems in compilation for parallel machines. Vectorizing compilers do not address the issues of data layout and communication. Questions related to compilation for distributed-memory machines are mostly still open.

The great promises and challenges of the compilation approach are the motivations of this dissertation.

1.2 Research Objectives and Approach

The objective of this dissertation is to identify key problems in compilation for distributed-memory machines and to seek practical solutions for them. Our ultimate goal is to have *automatic data layout* and *automatic generation of efficient communication.*

To this end, we choose to base our approach on the *data-parallel paradigm*, in which a large amount of data is addressed collectively. In our compiler system, the source language is the high-level functional language Crystal. It contains special constructs for expressing data parallelism. The target code generated by our compiler is in the so-called SPMD *(Single Program Multiple Data)* style, i.e. every processor of the target machine executes the same program. The conditionals in the program control what a specific processor does.

With respect to data layout over processors, a set of simple distribution strategies is considered. These strategies have a common feature that a data array is partitioned into sub-arrays that are more or less the same size and shape and are distributed evenly to the processors of the target machine.

To generate interprocessor communication, we consider not only the primitive send and receive commands, but also many powerful *aggregate communication routines* such as shifting, broadcasting, reduction, and array transpose.

1.3 Related Work

Research on compiling high-level languages for parallel machines can be traced back to the 70's. Kuck and his co-workers developed the Parafrase compiler, a restructuring Fortran compiler targeted for vector machines [PKL80,KLC+83,KKLW84,PW86]. The compiler performs comprehensive dependence analysis on the input sequential Fortran program to detect parallelism and then applies various transformations to generate a target program in vector form. Kennedy and his colleagues developed the PFC automatic translator, which converts Fortran programs to Fortran 8x programs [AK87]. The same group later developed the ParaScope Editor, which provides an interactive programming environment for parallelizing Fortran 77 programs [CCH+88,KMT91]. Advanced techniques for analyzing interprocedual data dependences have been built into the PFC and the ParaScope systems [CKT86]. Researchers at IBM Research at Yorktown Heights developed the PTRAN compiler system [ABC+86,Sar91], which made extensive use of control dependence to acieve better automatic parallelization.

Research efforts on compilation for distributed-memory machines started much later [CK88,CCL88,RS88,ZBG88], but ever since it has grown rapidly. A large number of researchers are building compilation systems or studying compilation techniques. The following is a brief review of them.

Callahan and Kennedy [CK88] studied techniques for compiling a version of Fortran 77 that includes annotations for specifying data decomposition for distributed-memory machines. They propose that the user provide data distribution functions and the compiler generate communication statements (*load* and *store* commands) for nonlocal

data references. They also propose a method for applying vectorization techniques to aggregate messages to reduce overhead. Their techniques can handle both simple doall loops and more involved doacross loops.

Fortran D Fortran D [FHK+90,HKK+91] is a version of Fortran enhanced with data decomposition specifications, proposed by Kennedy and his co-workers. The DECOMPOSITION statement is used to declare a problem domain for each computation; the ALIGN statement is used to specify how arrays should be aligned with respect to one another; and the DISTRIBUTE statement is used to map the problem and its associated arrays to the physical machine. The goal of Fortran D is to provide an efficient machine-independent parallel programming model. A Fortran D compiler for the iPSC/860 is under construction [HKT91a,HKT91b].

Fortran 90 Fortran 90 extends Fortran 77 with a set of parallel constructs and intrinsic functions. The parallel constructs support whole array operations and array sections, which simplifies the writing of data parallel applications. Wu and Fox proposes to compile Fortran 90 programs for distributed-memory MIMD machines [WF91]. The basic compilation strategy is to block the multidimentional array operations into submatrix operations with different submatrices assugned to different processors.

SUPERB Zima, Bast and Gerndt [ZBG88,Ger89] developed SUPERB, an interactive system for semi-automatic transformation of Fortran 77 programs into parallel programs for the SUPRENUM multiprocessor machine. The user of SUPERB specifies the mapping from data arrays to processors. The compiler performs dependence analysis and program transformations, and generates communications. The compilation techniques used in the SUPERB system are similar to those used in the Rice system. However, the SUPERB system can automatically compute the *overlaps* between the distributed array sections, and hence avoid locality-checks during execution.

DINO Rosing, Schnabel, and Weaver [RS88,RSW88,RSW90] developed DINO, a C-like, explicit parallel language. A construct called *environment* is used to represent an abstract parallel machine and *composite functions* are used to specify parallel computations. The DINO system requires the user to specify array distribution. It also requires the user to annotate nonlocal data references. The user can specify block, cyclic, and stencil-based data distributions. With the directives, the DINO compiler assigns processes to each processor and generates communications.

Kali Mehrotra, Van Rosendale and Koelbel [KM89,KMV90] developed Kali, a Fortran based language. The user of the Kali system provides the following information: 1) the processor array on which the program is to be executed, 2) the distribution of the data structures across these processors, and 3) the parallel loops and where they are to be executed. The Kali compiler automatically generates the communications. The Kali system also uses an *inspector/executor* strategy for processing communication at runtime [KMSB90a], which enables it to handle irregular computations.

Id Nouveau Rogers and Pingali [RP89] developed techniques for compiling the data flow language Id Nouveau on hypercube machines. They present a compilation system which, given a program and its data partition, performs task partitions and communication generation. The main technique is called *compile-time resolution* which calculates the set of *evaluators* and *participants* for each statement. Send and receive pairs for passing blocks of data between processors can be generated. The system uses a *runtime resolution* algorithm to handle dynamic messages at runtime.

CM FORTRAN CM FORTRAN [cmf89] is an implementation of FORTRAN 77 supplemented with array-processing extensions from the draft ANSI standard FORTRAN 8x. The user of CM FORTRAN explicitly specifies parallelism in his program via array assignments and the forall loop construct. The CM FORTRAN compiler distributes the data arrays and generates communications. It also automatically aligns arrays based

on the analysis of the usage patterns of array occurrences [KLS88,KLS90]. The compiler relies heavily on the underlining SIMD architecture of the Connection Machine, and hence does not address many issues that are critical to MIMD computers.

C* C* [RS87] is a C and C++ based language developed to support data parallel programming. The user specifies parallel computations as *actions* on a *domain.* The compiler automatically determines the data distribution and generates communications. Quinn and Hatcher [QHV88] studied the compilation of C* for distributed-memory machines. They developed optimization techniques for minimizing the number of *global synchronizations* and for reducing communication overhead.

AL AL [Tse89,Tse90] is a special array language for the WARP distributed-memory systolic array [KM84,AAG+87]. Parallel arrays are explicitly labeled in the user program. The AL compiler distributes one-dimensional arrays and generates communication automatically. The compiler does not aggregate messages because of the communication ability of the target machine.

PARTI and ARF PARTI [SCMB90] is a set of run-time library routines that support irregular computations on MIMD distributed-memory machines. ARF provides a Fortran interface for accessing PARTI run-time routines. ARF [KMSB90b,WSBH91] supports both regular distributions and user-defined irregular distributions. It generates *inspector* and *executor* loops for run-time processing.

A few brief comments on comparison of our compilation system and the above systems are given below. (1) Despite the variety of languages used in these compilation systems, the major forms of parallelism are parallel loops over arrays. Crystal has the same power in expressing parallelism. (2) In most cases, the user-provided data distribution directives in the above systems are used to specify regular distribution functions such as *block*, *cyclic*, and their combinations. In our systems, a framework for automating these special partitioning strategies is provided. (3) The forms of com-

munication generated by the above systems are variations of simple sends and receives. Our system is based on a different model. We extract syntactic reference patterns that represent communication from the source program, and match each of them with the most efficient communication routine. Using our technique, non-primitive communication forms such as broadcasting and reduction, can be generated.

1.4 Organization of the Dissertation

Chapter 2. The source language Crystal and the overall structure of the Crystal compiler are introduced. Intermediate program representations and an abstract machine model are also presented.

Chapter 3. An overview of key compilation techniques is given. These techniques facilitate two major program transformations, one from a Crystal program to a shared-memory parallel program and the other from a shared-memory program to a message-passing program. A general framework for optimizing the target code is also presented.

Chapter 4. The index domain alignment problem is defined and studied. The problem is modeled as a graph problem, and is shown to be NP-complete. Practical algorithms for solving the problem are presented and their performance is studied.

Chapter 5. The problem of deriving parallel control structures from a Crystal program is studied in detail. Issues addressed here include the representation of data dependence of a Crystal program, the correspondence between Crystal program constructs and shared-memory program constructs, the algorithm for synthesizing control structures, and the analysis of the algorithm.

Chapter 6. The generation of explicit communications is the focus of this chapter. An approach based on matching reference patterns with collective communication

routines is presented. Three major issues in communication generation—the synthesis, the synchronization, and the scheduling of communication primitives—are studied in detail.

Chapter 7. Benchmark results of the experimental Crystal compiler for a set of applications are presented and analyzed.

Chapter 8. Concluding remarks and directions of future work are presented.

Chapter 2

Language and Machine Models

This chapter introduces the source language Crystal and the compilation model of the Crystal compiler. Section 2.1 presents a brief description of Crystal, emphasizing the constructs that are most relevant to this dissertation. Section 2.2 defines the target machine model for the compiler. An overview of the structure of the compiler is given in Section 2.3. In the compilation process, a Crystal program is first transformed into a shared-memory program, then the shared-memory program is transformed into a message-passing program, and finally, the message-passing program is transformed into a program executable on a specific target machine. Section 2.4 describes the shared-memory program model. Section 2.5 describes data layout strategies. Finally, in Section 2.6, the message-passing program model is presented.

2.1 Crystal

Crystal [Che86] is a language designed to provide a convenient means for expressing parallelism and locality. It contains special constructs for representing data parallel computations, and powerful operators for expressing aggregate operations. Crystal is a functional language. It has clean semantics and nice algebraic properties. For instance, program transformations can be mathematically defined and mechanically

carried out [CC88]. We do not intend to describe the language in its full extent here. We will only introduce major Crystal features that are relevant to this dissertation.

Programs

A Crystal program contains a set of possibly mutually recursive definitions and optional output expressions. Definitions can be classified into index domain definitions, data field definitions, function definitions, and definitions of constants. Index domains and data fields are intended for representing distributed data structures and data parallel computations, and functions are intended for representing atomic sequential computations and combining data parallel components (via high-order functions). We give a brief description of each of the constructs below.

Index Domains

Index domains are abstractions of the "shapes" of composite data structures. An index domain consists of a set of elements, called *indices.*

An *interval domain*, denoted $[m..n]$, where m and n are integers and $m \leq n$, is one of the basic index domains. The elements of the interval domain are the set of integers $\{m, m+1, m+2, \ldots, n\}$. Other basic index domains include the *tree domain* and the *hypercube domain.*

Given two index domains D and E, we can construct their *product* $(D \times E)$ and *disjoint union* $(D + E)$ in the usual way.

In a Crystal program, index domains are defined by the keyword dom. For example, to define an n by n index domain D, we write

$$\mathsf{dom}\ D = [1..n] \times [1..n].$$

Data Fields

Data fields generalize the notion of distributed data structures, unifying the conventional notions of arrays and functions. A data field is a function over some index

domain D into some domain of values V. Usually, V will be the integers or the floating point numbers. V can also be some domain of data fields.

A parallel computation is specified by a set of data field definitions, which may be mutually recursive. The following example illustrates the use of data fields:

$$\begin{aligned}
&\textsf{dom } D_1 = [0..n] \\
&\textsf{dom } D_2 = D_1 \times D_1 \\
&\textsf{dfield } a(i) : D_1 = i + 2 \\
&\textsf{dfield } b(i,j) : D_2 = \textsf{if } (i = 0) \textsf{ then } a(j) \\
&\qquad\qquad\qquad\qquad \| \textsf{ else } b(i-1,j) + a(j) \\
&\qquad\qquad\qquad\qquad \textsf{fi}
\end{aligned}$$

D_1 is an interval domain and D_2 is a product domain. Data field a is defined over D_1 while data field b is defined over D_2.

Functions

Any function that is not explicitly defined over an index domain is just a conventional function. Functions are used for specifying computations that are intended to be executed sequentially. Functions can be self-recursive or mutually recursive. The following example illustrates the use of functions:

$$\begin{aligned}
&a(i) = i + 2 \\
&b(i,j) = \textsf{if } (i = 0) \textsf{ then } 0 \\
&\qquad\qquad \| \textsf{ else } b(i-1,j) + a(j) \\
&\qquad\qquad \textsf{fi}
\end{aligned}$$

These two functions will produce the same results as the two data fields in the previous example. The difference is that functions are not intended to be executed in parallel.

Conditionals

The general form for a conditional expression is

$$\textsf{if } p_1 \textsf{ then } \tau_1$$

|| p_2 **then** τ_2

...

|| **else** τ_n

fi

where the p_i's are boolean expressions called *guards*, and the τ_i's are arbitrary expressions. The guards p_i are mutually exclusive. The value of the conditional expression is τ_k if p_k is true.

Tuples and Aggregate Operators

Tuples are for expressing elements in aggregate forms. They are comparable to arrays in other languages. Below are examples of tuples.

$$a = [1, 2, 3, 4, 5, 6]$$
$$b = [[12, 2], [4, 7]]$$

Several APL-like *aggregate operators* are supported in Crystal. The reduction operator "\" takes as arguments a binary associative function $\oplus : T \times T \rightarrow T$ over some data type T and a tuple $a = [a_1, \ldots, a_n]$ of elements of type T, and is defined as

$$\backslash \oplus a = a_1 \oplus \cdots \oplus a_n.$$

For example, $\backslash + [1, 2, 3, 4, 5, 6] = 21$.

A Program Example

Figure 2.1 shows a Crystal program for computing the product of two matrices.

Remarks

In this dissertation, we focus on compilation techniques for basic parallel constructs of Crystal. We do not address the problem of compiling every feature of the Crystal

```
! a0 and b0 are input arrays
dom D = [1..n] × [1..n]
dfield a(i,j) : D = a0[i,j]
dfield b(i,j) : D = b0[i,j]
dfield c(i,j) : D = \ + {a(i,k) * b(k,j) | 1 <= k <= n}
```

Figure 2.1: MM: A Crystal program for matrix multiplication.

language. For instance, we do not consider complex index domains such as tree domains or hypercube domains, nor complex data fields such as those whose return values are other data fields (i.e. higher-level data fields). The excluded features either have no immediate relation to parallel computation or require different compilation techniques than those presented in this dissertation.

2.2 Target Machine Model

We define a simple model to represent a class of actual distributed-memory multiprocessors. The physical target machines under consideration include various hypercubes such as the iPSC/860 and the nCUBE 2, transputer arrays, and the iWarp systolic arrays.

An abstract target machine has two major functions. Firstly, it provides a base for the machine-independent message-passing program model. With the abstract target machine, data partitioning functions and communication routines can be defined. A message-passing program hence can be generated without referring to a specific target machine. Secondly, communication metrics reflecting the communication cost on an actual target machine can be defined using an abstract target machine. These metrics are critical in decision-making processes at various compilation stages.

2.2.1 Abstract Machine Definition

An abstract target machine is defined as a set of identical, independent processors configured as a multi-dimensional grid. The topology of the network connecting the processors is not explicitly specified. The communication capabilities of the machine are described by some communication metrics based on message-passing patterns.

An abstract machine can be represented by a pair $\mathcal{M} = [E, C]$ where

- $E = [1..n_1] \times \cdots \times [1..n_k]$ is an index domain in the form of a Cartesian product of k interval domains for some integer k. E represents the spatial configuration of the processors of the abstract machine, and will be referred to as the *processor domain.*

- C is a communication metric. The concrete form of C is flexible; it can either be a closed-form function or a table of costs.

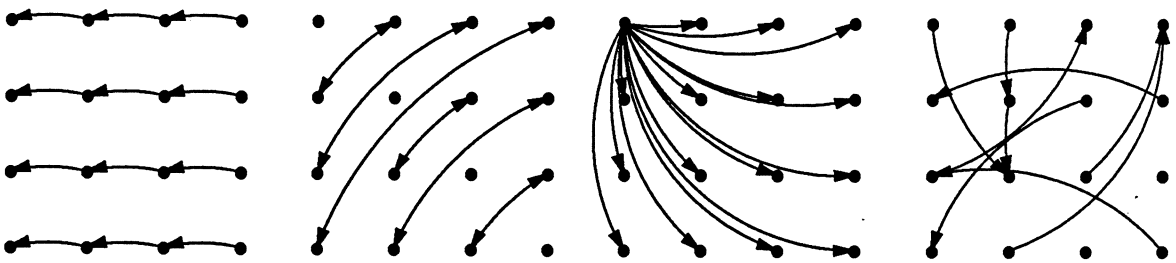
Multi-dimensional grid configuration is both simple and useful. Many data parallel application programs use multi-dimensional array as the major data structure, which maps naturally to multi-dimensional grids. In addition, multi-dimensional grids can be nicely embedded into many architectures, in particular, meshes and hypercubes.

2.2.2 Communication Patterns

Interprocessor communication on an abstract machine can have several forms:

- *Uniform Communication:* The relationship between the sender and the receiver is invariant of their locations. For example, every processor sends a message to its neighboring processor on the left, as shown in Figure 2.2(a).

- *Permutation:* A collection of processors participate in a loosely synchronous communication [FJL+88]. Each processor sends or receives at most one message. An example is that of transpose, i.e. sending data in a row of processors to a column of processors (Figure 2.2(b)).



(a) uniform (b) permutation (c) aggregate (d) arbitrary

Figure 2.2: Different forms of communication.

- *Aggregate Communication:* This form of communication is used to achieve either replication of data or reduction and scan of multiple data elements for a group of processors in a loosely synchronous manner. Figure 2.2(c) shows data on a processor being broadcast to all others.

- *Arbitrary Communication:* A collection of processors participate in a totally asynchronous communication. No particular characterization of the relation between the senders and receivers can be given, as shown by an example in Figure 2.2(d).

The following notation describes communication precisely.

Definition 2.1 Consider an n-dimensional processor domain E. Let $(\sigma_1, \ldots, \sigma_n)$ and $(\delta_1, \ldots, \delta_n)$ be two tuple expressions whose values range over E. We define a *collective communication* to be the collective data motion of moving data from $(\sigma_1, \ldots, \sigma_n)$ to $(\delta_1, \ldots, \delta_n)$ for all $(\sigma_1, \ldots, \sigma_n) \in E$. A collective communication can be described by a *communication pattern*, as follows

$$\ulcorner a@(\sigma_1, \ldots, \sigma_n) \Rightarrow (\delta_1, \ldots, \delta_n) : E \urcorner \tag{2.1}$$

Tuple $(\sigma_1, \ldots, \sigma_n)$ is called the *source expression*, and $(\delta_1, \ldots, \delta_n)$ the *destination expression*. Variable a represents the data to be transmitted.

The communication pattern of a collective communication has two special forms. In the *sender's form*, the source expression consists of index variables $(i_1, \ldots, i_n)$ only, which range over E. In the *receiver's form*, the destination expression consists of index variables only:

$$\text{Sender's form:} \quad \ulcorner a@(i_1, \ldots, i_n) \Rightarrow (\delta'_1, \ldots, \delta'_n) : E \urcorner,$$
$$\text{Receiver's form:} \quad \ulcorner a@(\sigma'_1, \ldots, \sigma'_n) \Rightarrow (i_1, \ldots, i_n) : E \urcorner.$$

When $(\sigma'_1, \ldots, \sigma'_n)$ and $(\delta'_1, \ldots, \delta'_n)$ are linear expressions of the indices, it is possible to symbolically convert between the sender's form and the receiver's form.

Communication patterns can be used to describe uniform communication, permutation, and aggregate communication. The notion can also be used to describe an arbitrary communication; but since an arbitrary communication assumes no data movement pattern, each sender and receiver pair must be described individually.

Example 2.1 Denote the processor domain in Figure 2.2 by $E = [1..4] \times [1..4]$ (assuming top left element is $(1,1)$). The four communication forms can be described as follows:

(a) $\ulcorner a@(i,j) \Rightarrow (i-1, j) : E \urcorner$;

(b) $\ulcorner a@(i,j) \Rightarrow (j, i) : E \urcorner$;

(c) $\ulcorner a@(1,1) \Rightarrow (i,j) : E \urcorner$;

(d) $\ulcorner a@(1,1) \Rightarrow (3,2) : E \urcorner$, $\ulcorner a@(1,2) \Rightarrow (2,2) : E \urcorner$, ...

Using the notion of communication pattern, we can define an important concept, *uniformity.*

Definition 2.2 A communication pattern

$$\ulcorner a@(i_1, \ldots, i_p, \ldots, i_n) \Rightarrow (\tau_1, \ldots, \tau_p, \ldots, \tau_n) : E \urcorner$$

is *uniform* in the pth dimension of E if $\tau_p \cong \ulcorner i_p + c \urcorner$, where c is a small constant independent of domain bounds, and $\cong$ denotes that two expressions have the same canonical form.[1]

Example 2.2 Communication pattern $\ulcorner a@(i,j) \Rightarrow (i-1,j) : E \urcorner$ is uniform in both dimensions of E, while pattern $\ulcorner a@(i,j) \Rightarrow (i-1,i+j) : E \urcorner$ is uniform only in the first dimension.

Communication patterns over a processor domain can be characterized according to their uniformity. Some patterns are uniform in all the dimensions, some are uniform in all but one dimension, some are uniform in only one dimension, and so forth.

Communication patterns over a processor domain can also be characterized according their "complexity". Patterns that can be implemented efficiently on an actual target machine are defined as "primitive patterns." Other communication patterns are then viewed as compositions of these primitive patterns. Note that a communication pattern with very low degree of uniformity may be a primitive pattern, for instance *one-to-all broadcasting.* A full discussion on primitive patterns is given in Section 6.2.

2.2.3 Communication Metrics

On an actual target machine, different communication patterns imply different source and destination relationships, different traffic patterns, and hence different communication costs. For instance, the following observations are generally true on machines with regular interprocessor network topology such as mesh or hypercube:

- *A local memory access is far faster than interprocessor communication.* The difference in cost can be as much as one or two orders of magnitude.

[1] A canonical form of an expression is a syntactic form in which variables appear in a predefined order and constants are partially evaluated. For example, $\ulcorner 2 - i + j \urcorner$ and $\ulcorner j - i + 3 - 1 \urcorner$ would have the same canonical form $\ulcorner -i + j + 2 \urcorner$.

- *The more uniform a pattern is, the lower is the communication cost.* Nonuniformity implies nonlocal communications that are likely to cause message collisions. The more nonuniform dimensions a pattern has, the greater the chance of message collisions.

The following two communication metrics are defined for our abstract target machine. They reflect different levels of information about an actual target machine.

Communication Matric 1: Classify communication patterns according to their uniformity and assign an appropriate cost to each class. This metric is based on uniformity of communication patterns. It relies on very little information about the target machine. It is suitable for use in early stages of the compilation.

Communication Matric 2: Compute the costs of a set of primitive communication patterns and then estimate the cost of general communication patterns by decomposing them into primitive patterns. This metric is based on a reduction rule. It is more refined, since the costs of the primitive patterns can be estimated quite accurately by actually implementing them on the actual target machine.

2.3 The Structure of the Compiler

Figure 2.3 shows the structure of the Crystal compiler. The compiler transforms a source Crystal program through several phases into a C program program augmented with communication statements. The compilation process consists of three major phases: (1) index domain alignment and control structure synthesis, (2) data partitioning and communication generation, and (3) code generation. The two intermediate representations in between the three phases are a shared-memory program and a message-passing program, respectively. The final output of the compiler is a

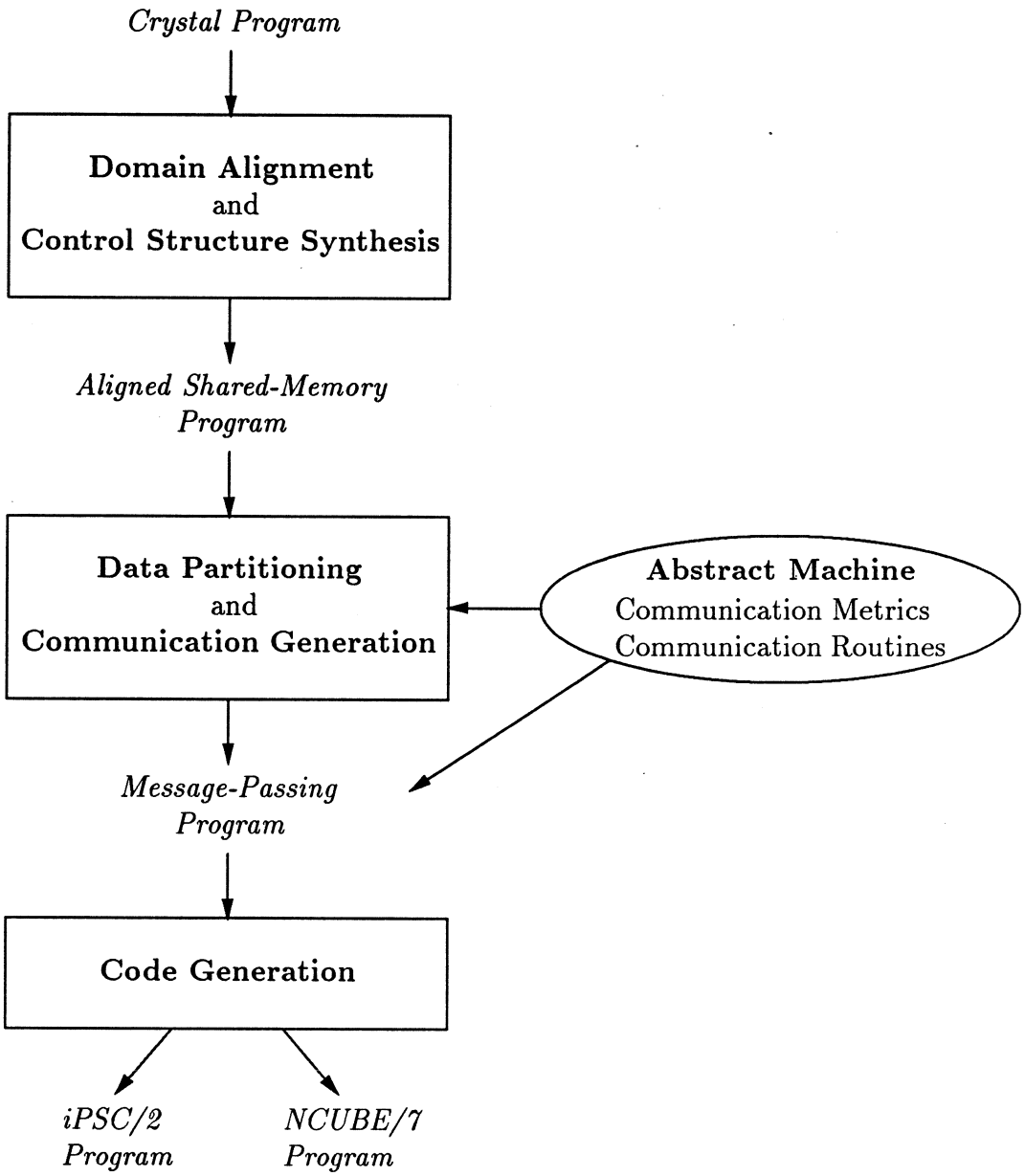


Figure 2.3: An overview of the Crystal compiler.

program suitable for compilation and execution on a specific target machine. In addition, an abstract target machine as defined in the previous section is used to provide guidance for compiler optimizations.

2.3.1 Index Domain Alignment and Control Structure Synthesis

The first major phase of the compilation, index-domain alignment and control-structure synthesis, transforms a Crystal program into an aligned shared-memory parallel program.

In this phase, the compiler performs data-dependence analysis on the source Crystal program to reveal the implicit parallelism and the implicit control structures of the program. The information derived is added to the source program through explicit constructs, transforming it into an explicitly parallel program.

The compiler also performs automatic domain alignment in this phase. The purpose of domain alignment is to determine the relative allocations of data fields that would minimize data movement between them. The major steps are as follows. The source program is first decomposed into segments called *blocks*, based on the dependence analysis result. Then dependences between data fields of the same block are analyzed and the result is represented by an undirected graph. The alignment algorithm is applied to the graph to generate alignment functions for the data fields. Finally, the alignment information is added to the source program through redefinitions of data fields. Hence, the resulting program from this phase is a shared-memory program with explicit alignment and parallelism.

2.3.2 Data Partitioning and Communication Generation

The second major phase of the compilation, data partitioning and communication generation, transforms an aligned shared-memory program into a message-passing program for an abstract target machine.

This phase deals with two most important issues in compiling programs for distributed-memory machines, data partitioning and interprocessor communication. We represent partitioning strategies by a set of symbolic parameters, and analyze communication with respect to these parameters. The result of the communication analysis is then used to select the best partitioning strategy.

Given a set of symbolic partitioning parameters, the compiler takes the following steps to generate interprocessor communication. Firstly, array references in the shared-memory program are collected and represented in a data structure called reference patterns. Secondly, each reference pattern is analyzed and a single or a combination of several communication routines from a set of predefined communication routines is selected. Optimizations are applied in the process so that the most efficient routines are selected. Next, communication routines generated from array references are inserted into the shared-memory program. Appropriate conditionals are attached to these communication routines to ensure synchronization of the message passing. Finally, a partitioning strategy is selected based on the result of communication analysis, and an integrated message-passing program is generated.

2.3.3 Code Generation

The last phase of the compilation generates the actual target code for a specific distributed-memory machine from an abstract message-passing program. The actual target code is supported by a runtime system implemented on the target machine. The runtime system consists of two libraries: one provides communication routines, the other provides index mapping routines for embedding an abstract machine and for implementing data partitioning.

2.4 Shared-Memory Parallel Programs

In this section, we describe a model of shared-memory parallel programs for representing explicit alignment information and parallelism.

2.4.1 Program Structure

A shared-memory program consists of a sequence of statements, separated by semi-colons (";"). The semi-colons indicate a sequential execution order. For example, the sequence $S_1; S_2$ means that statement S_1 is executed before statement S_2.

There are two restrictions that apply to a shared-memory program:

- *The single-assignment rule:* Each array element can be assigned a value only once. However, an array can appear on the left-hand side of many assignment statements (so long as different array elements are assigned values each time).
- *Restricted left-hand side subscripts:* Array subscript expressions on the left-hand side of an assignment statement must be index variables. For instance, the statement $a(i, j-1) = b(i+2, j)$ should be written as $a(i, j) = b(i+2, j+1)$.

These two restrictions can be naturally satisfied when transforming a Crystal program to a shared-memory program.

2.4.2 Loop Constructs

Three types of loops are used in our shared-memory programs: for, forall, and while-active. A for loop is just a sequential loop in which iterations are executed sequentially. The other two loops are described below.

Forall Loops

Syntactically, a forall loop is represented by a keyword forall followed by a declaration of the loop index and its iteration space (i.e. an interval domain) followed by the

loop body. For instance, **forall** $(i : [1..n])$ $S(i)$; represents a **forall** loop of n iterations. Nested **forall** loops can be represented by a combined header as in the following example:

$$\begin{array}{l}\textbf{forall } (i : D_1) \\ \quad \textbf{forall } (j : D_2) \\ \quad\quad S(i,j);\end{array} \quad \Longleftrightarrow \quad \begin{array}{l}\textbf{forall } ((i,j) : D_1 \times D_2) \\ \quad\quad S(i,j);\end{array}$$

The semantics of the **forall** loop is quite simple, due to the single-assignment rule.[2] A **forall** loop with a single array assignment statement represents a completely parallelizable loop—all its iterations are independent and hence can be executed in any order.

Multiple array assignment statements are allowed in a **forall** loop, provided that there is no cross-iteration dependences among the statements. Such a loop can always be distributed over the statements in its body, resulting in a collection of **forall** loops. For instance,

$$\begin{array}{l}\textbf{forall } (i : D) \ \{ \\ \quad a(i) = \alpha; \\ \quad b(i) = \beta; \\ \}\end{array} \quad \Longrightarrow \quad \begin{array}{l}\textbf{forall } (i : D) \\ \quad a(i) = \alpha; \\ \textbf{forall } (i : D) \\ \quad b(i) = \beta;\end{array}$$

While-Active Loops

A nonconventional loop, the **while-active** loop, is included in our model of shared-memory program. It is used when the compiler can derive neither a **for** loop nor a

[2]Without the single-assignment rule, there can be many ways to handle the problem of multiple writes to the same memory location. Hence many different semantics can be defined for **forall** loops or other equivalent parallel loops (see, for instance, [LB80,cmf89,Wol89]).

```
dom D = [1..n] × [1..n];
forall((i, j) : D)
  a(i, j) = a0[i, j];
forall((i, j) : D)
  b(i, j) = b0[i, j];
forall((i, j) : D)
  c(i, j) = \ + {a(i, k) * b(k, j) | 1 <= k <= n};
```

Figure 2.4: The shared-memory version of the MM program.

forall loop for a group of data fields due to the lack of information. In such a case, these data fields are put into a while-active loop.

The semantics of a while-active loop

$$\textsf{while-active } (i : D)$$
$$a(i) = \alpha[i, a];$$

is as follows. Compute $\alpha[i, a]$ for every possible i in D. If some i, $\alpha[i, a]$ is dependent on some elements of a, the computation would be switched to other i's. Repeat this process until all the values of a are computed or a deadlock is reached.

On a sequential computer, the while-active loop can be emulated by a while loop: keep computing while there are still "active" (i.e. computable) elements of a. To execute a while-active loop in a multiprocessor environment, appropriate run-time support is needed. First of all, each processor must be able to handle interruptions, since a message may arrive at any time. Secondly, there must be a queuing manager which delays requests until the data is available. The overhead associated with the execution of a while-active can be very high.

Example 2.3 Figure 2.4 shows the shared-memory version of the matrix multiplication program.

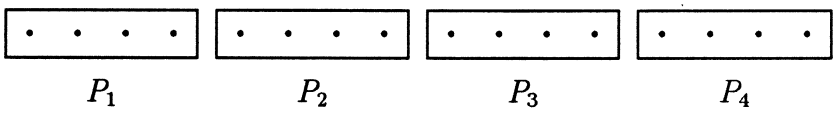


Figure 2.5: A 4-block layout of a one-dimensional domain.

2.5 Data Layout Strategies

We use the term *data layout* to refer to the process of decomposing an index domain into segments and mapping the segments onto the processor domain of an abstract target machine. Since our programs are based on the data-parallel model, decomposing the index domain of an array implies that both the data and the computation associated with the array are decomposed.

Data layout consists of two issues: *partitioning a domain* and *distributing the domain partitions.* We consider these two issues together, since the distribution of the domain partitions to a processor domain is straightforward under our abstract machine model—both the array index domain and the processor domain are Cartesian products of intervals and we can adjust the parameters of the abstract machine so that there is always a simple relationship between the two.

In the Crystal compiler, two data layout strategies are considered. Both strategies decompose an index domain into subdomains of the same size and shape.

The Block Layout Strategy

Definition 2.3 Given an interval domain $D = [1..n]$, a *k-block layout strategy* divides D into k equal-size segments and assigns them to an array of k processors, with the ith segment assigned to the ith processor (Figure 2.5).

In case k divides n, the k segments can be specified as

$$[1..n/k], [n/k+1..2n/k], \ldots, [(k-1)n/k+1..n]. \tag{2.2}$$

If k does not divide n, there are two standard ways to handle it. One way is to let the first l ($l < k$, its exact value is defined below) segments to be of the same size, $\lceil n/k \rceil$, and let the $(l+1)$th segment be of a smaller size $n - l\lceil n/k \rceil$. The value of l is given by

$$l = \lfloor n/\lceil n/k \rceil \rfloor. \tag{2.3}$$

The second approach is to let the first l segments to be of the size $\lfloor n/k \rfloor + 1$, and the remaining segments to be of size $\lfloor n/k \rfloor$. The value of l is given by

$$l = n - \lfloor n/k \rfloor. \tag{2.4}$$

The second approach guarantees that segments differ in size by at most one.

Example 2.4 Consider a domain $D = [1..41]$ and an array of 8 processors. Using the first approach, the layout is

$$6 \quad 6 \quad 6 \quad 6 \quad 6 \quad 6 \quad 5 \quad 0,$$

and the last processor is wasted. Using the second approach, the layout is

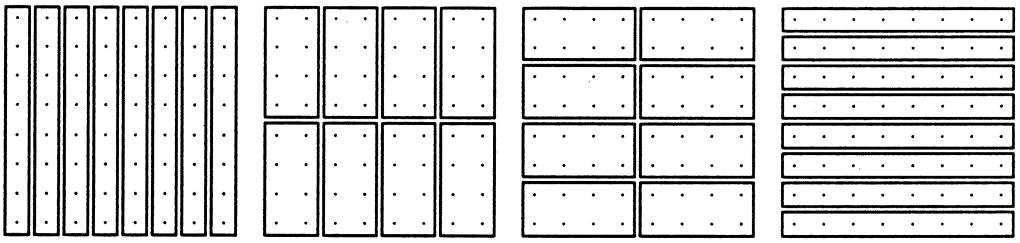
$$6 \quad 5 \quad 5 \quad 5 \quad 5 \quad 5 \quad 5 \quad 5,$$

which achieves maximal load balance.

The block-layout strategy can be applied to two-dimensional index domains as well. However, since a two-dimensional domain can be divided along either of the two dimensions (or both), we need two parameters to describe a specific layout.

Definition 2.4 A (k_1, k_2)-block layout strategy for a two-dimensional domain D divides the domain into $k_1 \times k_2$ segments and maps them to a $k_1 \times k_2$ processor domain. The segment indexed by (i, j) is assigned to processor (i, j), for all $1 \leq i \leq k_1; 1 \leq j \leq k_2$.

Example 2.5 Figure 2.6 shows four different layouts of a two-dimensional domain using the block-layout strategy.



(a) $(1,8)$-block (b) $(2,4)$-block (c) $(4,2)$-block (d) $(8,1)$-block

Figure 2.6: Block layouts of a two-dimensional domain.

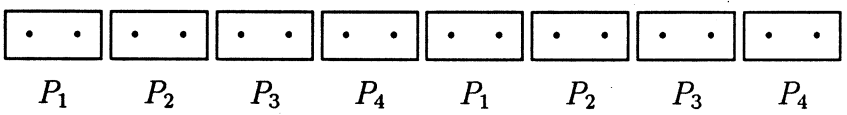


Figure 2.7: A $(4,2)$-interleaving layout of a one-dimensional domain.

The Interleaving Layout Strategy

Definition 2.5 A (k,l)-interleaving layout strategy for a one-dimensional domain D divides the domain into $k \times l$ segments and maps them to a k processor array. The ith, $(i+k)$th, $(i+2k)$th, ..., $(i+(l-1)k)$th segments are assigned to the ith processor.

Example 2.6 Figure 2.7 shows a $(4,2)$-interleaving layout strategy.

The advantage of an interleaving layout over a block layout is that it tends to balance the computation load more. However, the overhead of an interleaving layout is higher due to the context switching between segments assigned to the same processor. Furthermore, using an interleaving layout may increase communication cost, since messages tend to be smaller in size but larger in quantity.

A k-block layout strategy is a special case of a (k,l)-interleaving layout strategy where $l = 1$. In general, a data layout strategy for an n-dimensional index domain can be described by n pairs: $((N_1,l_1),\ldots,(N_n,l_n))$.

Notation

For the rest of the dissertation, the following notational convention is used. $\mathcal{P}$ denotes a data layout. $E(\mathcal{P}, D)$ denotes the processor domain that is the result of applying $\mathcal{P}$ to index domain D. $\hat{D}(\mathcal{P}, \boldsymbol{p})$ denotes the domain segment assigned to processor $\boldsymbol{p}$ (an element of $E(\mathcal{P}, D)$). When the context is clear, we simply use E and $\hat{D}$ to represent a process domain and an index domain segment.

2.6 Message-Passing Programs

In this section, we describe a model of message-passing programs for the abstract machine defined in Section 2.2. The message-passing program model uses the same program constructs (e.g. **forall** loops) as the shared-memory program model does. The data in a message-passing program, however, is now mapped to the private memory of the processors and communication statements are presented.

A message-passing program in our model is in the so-called SPMD (Single-Program-Multiple-Data) style. The program consists of an overall **forall** loop over the processor domain.[3] The body of the **forall** loop is to be executed on every processor. It consists of both **for** and **forall** loops, not necessarily in a perfectly-nested fashion.

Message-passing programs generated by the Crystal compiler are initially in a special form in which all **for** loops appear outside of **forall** loops. Figure 2.8 illustrates the general form of such programs. For simplicity, only one **for** loop is shown. The body of the **for** loop consists of a sequence of program segments: computation segments and communication segments. The structures of these two types of segments are also shown in Figure 2.8.

[3]To be precise, a different loop construct should be used for the overall loop, since its iterations are not completely independent.

```
forall (p : E(P, D))
  for (t : T) {
    ⟨Computation Segment⟩;
    ⟨Communication Segment⟩;
    ⟨Computation Segment⟩;
    ⟨Communication Segment⟩;
    ...;
  }
```

⟨Computation Segment⟩ ::

```
if pred then
  forall (i : D̂) {
    ⟨Index Mapping⟩;
    ⟨Array Assignment Statement⟩;
  }
```

⟨Communication Segment⟩ ::

```
if pred then {
  ⟨Buffer Preparation⟩;
  ⟨Communication Routine⟩;
  ⟨Data Rearrangement⟩;
}
```

Figure 2.8: The general program structure of a message-passing program.

2.6.1 Computation Segments

Consider a multiloop nest $\mathcal{L}$ of a shared-memory program that ranges over an index domain $D_1 \times D_2 \times \cdots \times D_n$. Assume that the first k loops of $\mathcal{L}$ are for loops (which range over domain $D_1 \times \cdots \times D_k$) while the inner loops from level $k+1$ to level n (denoted $L_{k+1}, \ldots, L_n$) are forall loops. Let $T = D_1 \times \cdots \times D_k$ and $D = D_{k+1} \times \cdots \times D_n$.

Given a data layout function $\mathcal{P}$ for D, the forall loops are eventually distributed over the processor domain $E(\mathcal{P}, D)$. Each processor gets assigned a portion of these loops. For a processor $\boldsymbol{p}$, the portion is over a subdomain $\hat{D}(\mathcal{P}, \boldsymbol{p})$.

We call a forall loop nest over a subdomain $\hat{D}(\mathcal{P}, \boldsymbol{p})$ in a message-passing program, a *computation segment.* For the above loop nest $\mathcal{L}$, a *computation segment* would appear in the form of

```
forall ((i_{k+1}, ..., i_n) : D̂(P, p)) {
  ⟨Index Mapping⟩;
  ⟨Array Assignment Statement⟩;
}
```

Although the forall loops in a message-passing program are to be executed sequentially by a single processor, we still denote them with the header forall. The reason is that the absence of data dependence of a forall loop allows communication to be scheduled in a way that better performance can be obtained.

2.6.2 Communication Segments

Since the iterations of a forall loop are fully independent, no communication has to occur inside a computation segment if the data it needs is fetched before the execution begins. However, between two different computation segments there might be data dependences, hence communication might be needed.

We call a group of statements in a message-passing program responsible for implementing a reference pattern a *communication segment.* Each communication segment

is guarded by a predicate that synchronizes multiple parties in a communication. It consists of statements for preparing a message buffer, a call to a communication routine, and statements for a possible rearrangement of the transferred data.

2.6.3 Index Mapping

In a message-passing program, each processor computes with local arrays that are the results of data partitioning on the original arrays. Some portions of these arrays (e.g. the temporal dimensions) can be accessed in the same way as the original arrays, but other portions must be accessed by using a different set of indexing variables.

The following three generic mapping functions are used to translate between three sets of indexing variables: (1) global indices for indexing portions of arrays that are not partitioned; (2) local indices for indexing the partitioned portions of arrays; and (3) processor indices for indexing the process domain. In all these functions, $\mathcal{P}$ denotes a data layout function, D denotes an index domain, $\boldsymbol{p}$ denotes a processor in the domain $E(\mathcal{P}, D)$, and $\boldsymbol{i}$ and $\hat{\boldsymbol{i}}$ denote an element in D and $\hat{D}(\mathcal{P}, \boldsymbol{p})$, respectively.

Global_to_Local($\mathcal{P}$, D, $\boldsymbol{i}$, $\boldsymbol{p}$): This function returns an element $\hat{\boldsymbol{i}} \in \hat{D}(\mathcal{P}, \boldsymbol{p})$ that corresponds to $\boldsymbol{i}$.

Local_to_Global($\mathcal{P}$, D, $\hat{\boldsymbol{i}}$, $\boldsymbol{p}$): This function returns an element $\boldsymbol{i} \in D$ that corresponds to $\hat{\boldsymbol{i}}$.

Index_to_Pid($\mathcal{P}$, D, $\boldsymbol{i}$): This function returns an element $\boldsymbol{p} \in E(\mathcal{P}, D)$ that holds the portion of D that contains the element $\boldsymbol{i}$.

On an actual target machine, each of these functions may be implemented by a group of routines.

As illustrated above, we use different variables for indexing different data: (1) plain variables like i and j are used to access global arrays; (2) variables with a hat, such as $\hat{\imath}$ and $\hat{\jmath}$, are used to access local arrays; (3) variables p and q are used to identify processors. Corresponding boldface letters are short forms of vectors.

```
dom D = [1..n] × [1..n];
dom E = E(D, P);
forall((p, q) : E) {
  forall((î, ĵ) : D̂)
    â(î, ĵ) = â0[î, ĵ];
  Multi_Spread(E, 1, â, a1, |D̂|);
  forall((î, ĵ) : D̂)
    b̂(î, ĵ) = b̂0[î, ĵ];
  Multi_Spread(E, 2, b̂, b1, |D̂|);
  forall((î, ĵ) : D̂)
    ĉ(î, ĵ) = \ + {a1(î, k) * b1(k, ĵ) | 1 <= k <= n};
}
```

Figure 2.9: The message-passing version of the MM program.

Example 2.7 Figure 2.9 shows the message-passing version of the matrix multiplication program. Note that references to data outside the processor's private memory are replaced with references to message buffers (e.g. a_1 and b_1) and a message buffer is filled as the result of communication.

Chapter 3

Compilation Strategies

In this chapter, we discuss the major compilation techniques for transforming a Crystal program into a message-passing program. Section 3.1 describes the first part of the transformation: from a Crystal program to a shared-memory parallel program. Section 3.2 describes the second part: from a shared-memory program to a message-passing program. Issues of compiling a whole program are addressed in Section 3.3. Finally, in Section 3.4, several code refinement techniques are presented.

3.1 Generation of Shared-Memory Programs

Crystal is a functional language. A Crystal program merely declares the index domains and specifies the definitions of data fields. Other kinds of information such as parallelism and control structures are implicit. The first phase of the Crystal compiler is to reveal the implicit parallelism of a Crystal program and to add control structures to the program.

Our method consists of the following steps (Figure 3.1). Firstly, data dependences between data fields and those between data values are analyzed. The results become the foundation for all the subsequent compilation analysis. Secondly, a Crystal program is decomposed into program segments called *blocks*. A program block is used

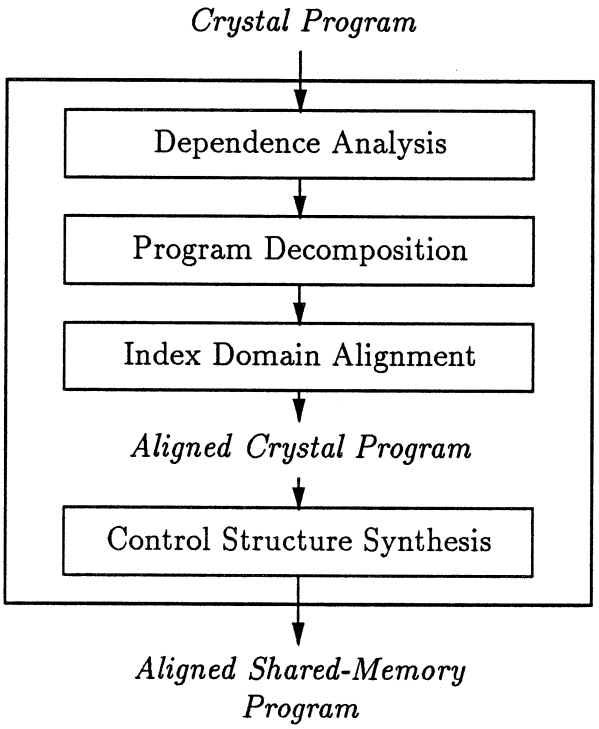


Figure 3.1: Generation of a shared-memory program.

as the standard unit in many compilation modules. Then, data fields belonging to the same program block are *aligned* to a common index domain. As the result, dependences between data fields can now be addressed in a unified framework, which enables the compiler to derive control structures more efficiently. Finally, each common index domain is analyzed with regard to data dependences. Some components are labeled *spatial* and others *temporal*. For each program block, loop nests containing both sequential and parallel loops are generated at the end. In the following, we describe these steps in turn.

3.1.1 Dependence Analysis

The purpose of dependence analysis is to reveal dependence relations between data objects in a Crystal program and represent them in some convenient form. For our purpose, the most important dependence relations are those between data fields.

In the following, the Greek letters, such as τ and ϕ, denote arbitrary Crystal expressions. A Greek letter with square bracket, such as $\tau[x_1, \ldots, x_n]$, denotes an expression containing the specifically given subexpressions.

Data Dependences

The functional nature of the Crystal language makes the dependence analysis of a Crystal program quite straightforward. There exists only one simple form of dependence in a Crystal program: data dependence, i.e. a value a depends on another value b. Hence, the compiler can derive dependence information by analyzing both sides of a definition equation.

We analyze data dependence of a Crystal program at two levels: (1) dependence between data fields, and (2) dependence between data values.

Definition 3.1 Consider the definition of a data field a:

$$\mathsf{dfield}\ a(\boldsymbol{i}) : D = \tau_1[b(\tau_2[\boldsymbol{i}])], \tag{3.1}$$

in which a data field b is referenced. We say that data field a is *call-dependent* on data field b.

Definition 3.2 With respect to the definition in Equation (3.1), for an specific element $\boldsymbol{i}_0 \in D$, the value $a(\boldsymbol{i}_0)$ is said to be *data-dependent* on the value $b(\tau[\boldsymbol{i}_0])$ if it is contained in the expression $\tau_1|_{\boldsymbol{i}=\boldsymbol{i}_0}$.

Note that a data field a can be call-dependent on itself, but a value $a(\boldsymbol{i})$ cannot be data-dependent on itself.

Example 3.1 Consider the following definition:

$$\begin{aligned}&\textsf{dom } D = [1..10] \times [1..10]\\ &\textsf{dfield } a(i,j) : D = \textsf{if } (i < 5) \textsf{ then } b(i,j)\\ &\qquad\qquad\qquad\qquad \mid\mid \textsf{else } a(i-1,j)\\ &\qquad\qquad\qquad\qquad \textsf{fi}\end{aligned}$$

Since references to both data fields a and b appear in the definition, data field a is call-dependent on both b and itself. For some indices $(i,j) \in D$, the value $a(i,j)$ is data-dependent on $b(i,j)$, while for others it is not. For instance, $a(1,1)$ is data-dependent on $b(1,1)$; but $a(10,10)$ is not dependent on $b(10,10)$, since when $(i,j) = (10,10)$, the right-hand expression becomes $a(9,10)$.

Call-Dependence Graphs

Call dependence derived from a Crystal program can be represented by a directed graph called *call-dependence graph* (CDG). A CDG can be defined for a whole program or for a subset of its data fields.

Definition 3.3 A *call-dependence graph* G for a group of data fields $\mathcal{S}$ is a directed graph. Each node of G corresponds to a data field in $\mathcal{S}$ and each directed edge corresponds to a call dependence between two data fields. The direction of the edge is defined as follows: if data field a is call-dependent on data field b, then the edge points from node b to node a.

Example 3.2 Consider the following data fields:

$$\begin{aligned}&\textsf{dfield } a(i,j) : D_1 = \tau_1[b(\phi_1[i,j])],\\ &\textsf{dfield } b(i,j) : D_2 = \tau_2[c(\phi_2[i,j])],\\ &\textsf{dfield } c(i,j,k) : D_3 = \tau_3[a(\phi_3[i,j,k]), c(\phi_4[i,j,k])],\\ &\textsf{dfield } d(x,y) : D_4 = \tau_4[d(\phi_5[x,y])],\end{aligned}$$

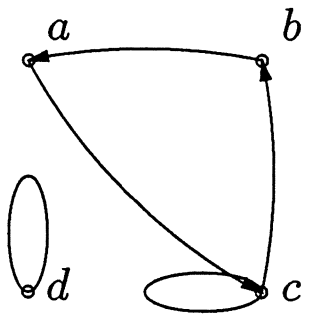


Figure 3.2: An example of call-dependence graphs.

Data field a is call-dependent on data field b. Data field b is call-dependent on c. Data field c is call-dependent on a and itself. And finally, d is call-dependent on itself. The CDG for these data fields is shown in Figure 3.2.

Reference Patterns

Data dependence at the level of data value is very fine-grained. Representing and analyzing each data dependence individually would require too much storage and processing time. We therefore define an aggregate representation form, called a *reference pattern.*

Definition 3.4 Consider the definition of a data field a. Assume it contains a reference to data field b as follows:

$$\mathsf{dfield}\ a(i_1, \ldots, i_n) : D = \tau[b(\tau_1, \ldots, \tau_n)].$$

The symbolic form

$$\ulcorner a(i_1, \ldots, i_n) \leftarrow b(\tau_1, \ldots, \tau_n) \urcorner \tag{3.2}$$

is called a *reference pattern* (the arrow $\leftarrow$ reads "is dependent on"). If $b(\tau_1, \ldots, \tau_n)$ appears in a conditional branch guarded by an expression γ, then the reference pattern takes the form of

$$\ulcorner a(i_1, \ldots, i_n) \leftarrow b(\tau_1, \ldots, \tau_n) : \gamma \urcorner. \tag{3.3}$$

A reference pattern of form (3.2) represents a collection of dependences: those between values $a(i_1, \ldots, i_n)$ and $b(\tau_1, \ldots, \tau_n)$ for all the elements in the index domain of a. Similarly, a reference pattern of form (3.3) represents dependences between values $a(i_1, \ldots, i_n)$ and $b(\tau_1, \ldots, \tau_n)$ for all the elements in the index domain of a for which the guard expression γ is true.

Example 3.3 Consider the following data field definition:

$$\textsf{dfield } a(i,j) : D = a(i, j-1) + b(j, i+1)$$

Two reference patterns, $\ulcorner a(i,j) \leftarrow a(i,j-1) \urcorner$ and $\ulcorner a(i,j) \leftarrow b(j,i+1) \urcorner$ can be derived.

3.1.2 Program Decomposition

Potentially, a Crystal program can be very large. It would be impractical to analyze and transform a large program as a single object. The purpose of program decomposition is to decompose a Crystal program into smaller and more manageable program segments.

We borrow the idea of the π-block used in decomposing Fortran programs[Kuc78]. We define a Crystal *block* to represent a group of data fields that are closely related to each other (the precise definition given below). Each block is treated as a single unit in analysis and transformation modules throughout the compiler. At the end the individually transformed blocks are linked back together to form an integrated target program.

Recall that a data field a is said to be call-dependent on a data field b if a reference to b appears in the definition of a. In order to formulate a Crystal block, we extend the dependence relation to data fields that are not directly call-dependent to each other.

Definition 3.5 A data field a is said to be *dependent* on a data field b if (1) a is call-dependent on b; or (2) there exists a data field c ($\neq a, \neq b$), a is call-dependent on c and c is dependent on b.

Algorithm Program_Decomposition(P)

Input: A Crystal program.
Output: A set of program blocks.

begin

Form the call-dependence graph of P;

Decompose the graph into strongly connected components;

Derive blocks based on the connected components.

end.

Figure 3.3: The program decomposition algorithm.

Definition 3.6 A Crystal *block* is a maximal set of data fields that are mutually dependent on each other.

With respect to the call-dependence graph of a Crystal program, a block corresponds to a strongly connected component of the graph. This correspondence provides a simple algorithm for computing blocks from a given program, as shown in Figure 3.3.

Example 3.4 The CDG in Figure 3.2 consists of two strongly connected components: one consists of nodes a, b and c, the other is a singleton d. Correspondingly, the four data fields can be decomposed into two blocks: $\{a, b, c\}$ and $\{d\}$.

3.1.3 Index Domain Alignment

Data fields of a Crystal program are defined over individual index domains. These domains may have the same shape and size, but otherwise are independent of each other. However, it is not always possible, nor desirable, to analyze and transform data fields individually, because data fields may be mutually dependent on each other and hence they may need to be transformed into a single loop nest.

Our approach is to analyze together all the data fields that belong to the same Crystal program block (by definition, they are mutually dependent). The index domain alignment step aligns the index domains of these data fields into a single common domain. By doing so, data field references, both self-references and cross-references, reside in a single reference framework. Further analysis can thus be based on conventional representations such as dependence vectors.

We formulate the domain alignment problem as finding a set of alignment functions that map a group of index domains into a common domain. There are many issues involved in this process. For instance, what type of alignment functions to consider? How to select alignment functions for a given index domain? And so on. Chapter 4 discusses the domain alignment problem in detail.

Example 3.5 Consider the following two data fields:

$$\begin{aligned}
&\mathsf{dom}\ D_1 = [1..n] \times [1..n] \\
&\mathsf{dom}\ D_2 = [1..n] \times [1..n] \times [1..n] \\
&\mathsf{dfield}\ a(i,j) : D_1 = a(i, j-1) + b(1,i,j) \\
&\mathsf{dfield}\ b(i,j,k) : D_2 = a(i+j, k-1)
\end{aligned}$$

The self-references $\ulcorner a(i,j) \leftarrow a(i,j-1) \urcorner$ can be analyzed with respect to the index domain D_1. The cross-references $\ulcorner a(i,j) \leftarrow b(1,i,j) \urcorner$ and $\ulcorner b(i,j,k) \leftarrow a(i+j,k-1) \urcorner$ however, involve two index domains. Without knowing the relationship between D_1 and D_2, little can be analyzed from the cross-references.

Suppose we aligned D_1 and D_2 to a common domain $D = [1..n] \times [1..n] \times [1..n]$ by the following alignment functions:

$$\begin{aligned}
&a(i,j) : D_1 \mapsto \tilde{a}(1,i,j) : D, \\
&b(i,j,k) : D_2 \mapsto \tilde{b}(i,j,k) : D.
\end{aligned}$$

Then the data fields can be transformed into

$$\mathsf{dom}\ D = [1..n] \times [1..n] \times [1..n]$$

$$\text{dfield } \tilde{a}(x,y,z) : D = \tilde{a}(x,y,z-1) + b(1,y,z)$$
$$\text{dfield } \tilde{b}(x,y,z) : D = \tilde{a}(1,x+y,z-1)$$

In this new form, all the data dependences can be represented with respect to the common domain D.

Note that in the process that transforms a Crystal program into a shared-memory program, the index domain alignment module works as a preprocessing step for the control structure synthesis module. However, the full effect of domain alignment goes far beyond that, and the details are elaborated in Chapter 4.

3.1.4 Control Structure Synthesis

The function of the control structure synthesis module is to make explicit both the sequential and the parallel control information in a Crystal program. It completes the transformation of a Crystal program into a shared-memory program.

The techniques for synthesizing control structures rely on natural correspondences between program constructs of Crystal and those of the shared-memory program model. In particular, they rely on the correspondences between index domains and loop iteration spaces, and those between data fields and multiloop nests. The central idea is to analyze data dependences with respect to a multidimensional index domain, to determine which dimensions are parallelizable and which dimensions must be computed sequentially, and finally, to construct parallel or sequential loops correspondingly.

There are several challenging issues involved in the synthesis process. First of all, for a given Crystal program more than one shared-memory program could be generated. Making selections among multiple outputs is not trivial, since the compiler does not have much information at this stage to provide accurate estimation of computation and communication overhead. Secondly, the correspondences between data fields and multiloop nests may not always be straightforward. The resulting loop

$A_0 = [[2.0, 1.0, 3.0, 4.0, 29.0], [1.0, 0.0, 0.0, 1.0, 5.0],$
$\quad [3.0, 1.0, 1.0, 0.0, 8.0], [5.0, 2.0, 0.0, 1.0, 13.0]]$
$n = 4$
dom $D_1 = [1..n] \times [1..(n+1)] \times [0..n]$
dom $D_2 = [1..n]$
dom $D_3 = [1..n] \times [1..n]$
dfield $a(i,j,k) : D_1 =$ if $(k = 0)$ then $A_0[i,j]$
$\quad$ || $(i < k)$ then $a(i,j,k-1)$
$\quad$ || $(i = k)$ then $a(ipivot(k),j,k-1)$
$\quad$ || $(i = ipivot(k))$ then
$\quad\quad a(k,j,k-1) - a(i,j,k-1) * fac(i,k)$
$\quad$ || else $a(i,j,k-1) - a(ipivot(k),j,k-1) * fac(i,k)$
$\quad$ fi
dfield $apivot(k) : D_2 =$ \max $\{|a(i,k,k-1)| \mid k <= i <= n\}$
dfield $ipivot(k) : D_2 =$ \max $\{i \mid k <= i <= n : |a(i,k,k-1)| = apivot(k)\}$
dfield $fac(i,k) : D_3 =$ if $(i <= k)$ then 1.0
$\quad$ || $(i = ipivot(k))$ then
$\quad\quad a(k,k,k-1)/a(ipivot(k),k,k-1)$
$\quad$ || else $a(i,k,k-1)/a(ipivot(k),k,k-1)$
$\quad$ fi

Figure 3.4: Gauss: A Crystal program for Gaussian elimination with partial pivoting.

nests may contain complicated nesting structures. Finally, there may exist Crystal programs whose dependences are too complex to analyze. Some default solutions need to be provided in case the compiler cannot synthesize any control structures from data dependences.

The control structure synthesis algorithm, the proof of its correctness and other related issues are presented in Chapter 5.

3.1.5 A Program Example

Figure 3.4 shows a Crystal program implementing an algorithm for Gaussian elimination with partial pivoting. The program iterates over the columns of the input matrix. In iteration k a pivot element is chosen from the elements in column k at or below the diagonal (say element (j, k)) and rows k and j are exchanged. Then, the elements in the column below the diagonal are eliminated using the pivot element. The aligned version and the shared-memory version of the program are shown in Figure 3.5 and Figure 3.6.

3.2 Generation of Message-Passing Programs

In this section, we give an overview of the transformation of a shared-memory program into a message-passing program. The details of the communication generation techniques of this transformation are presented in Chapter 6.

The following are two major issues involved in transforming a shared-memory program into a message-passing program:

- *Data layout:* Parallel computations specified by forall loops in a shared-memory program are partitioned and distributed over the processors of an abstract target machine.

- *Communication generation:* Communication statements for passing messages between processors are generated and inserted into the appropriate locations in the program. Since our target program is in the SPMD style, each communication statement must be guarded by appropriate conditionals.

These two issues are interdependent. To analyze communication, a data layout strategy must be given, because data distribution determines whether a communication is needed and if so, where the source and the destination are. On the other hand, the compiler does not know which data layout strategy is better until its cost can be estimated by determining the communication involved.

$A_0 = [[2.0, 1.0, 3.0, 4.0, 29.0], [1.0, 0.0, 0.0, 1.0, 5.0],$
$[3.0, 1.0, 1.0, 0.0, 8.0], [5.0, 2.0, 0.0, 1.0, 13.0]]$

$n = 4$

dom $D = [1..n] \times [1..(n+1)] \times [0..n]$

dfield $\tilde{a}(i,j,k) : D =$ if $(k = 0)$ then $A_0[i,j]$
|| $(i < k)$ then $\tilde{a}(i,j,k-1)$
|| $(i = k)$ then $\tilde{a}(\widetilde{ipivot}(1,k,k),j,k-1)$
|| $(i = \widetilde{ipivot}(1,k,k))$ then
$\tilde{a}(k,j,k-1) - \tilde{a}(i,j,k-1) * \widetilde{fac}(i,k,k)$
|| else $\tilde{a}(i,j,k-1) - \tilde{a}(\widetilde{ipivot}(1,k,k),j,k-1) * \widetilde{fac}(i,k,k)$
fi

dfield $\widetilde{apivot}(i,j,k) : D =$
if $((k > 0)$ and $(i = 1)$ and $(j = k))$ then
\max $\{|\tilde{a}(i,k,k-1)| \mid k <= i <= n\}$
fi

dfield $\widetilde{ipivot}(i,j,k) : D =$
if $((k > 0)$ and $(i = 1)$ and $(j = k))$ then
\max $\{i \mid k <= i <= n : |\tilde{a}(i,k,k-1)| = \widetilde{apivot}(1,k,k)\}$
fi

dfield $\widetilde{fac}(i,j,k) : D =$
if $((k > 0)$ and $(j = k))$ then
if $(i <= k)$ then 1.0
|| $(i = \widetilde{ipivot}(1,k,k))$ then
$\tilde{a}(k,k,k-1)/\tilde{a}(\widetilde{ipivot}(1,k,k),k,k-1)$
|| else $\tilde{a}(i,k,k-1)/\tilde{a}(\widetilde{ipivot}(1,k,k),k,k-1)$
fi
fi

Figure 3.5: The aligned Crystal version of the Gauss program.

```
A_0 = [[2.0, 1.0, 3.0, 4.0, 29.0], [1.0, 0.0, 0.0, 1.0, 5.0],
       [3.0, 1.0, 1.0, 0.0, 8.0], [5.0, 2.0, 0.0, 1.0, 13.0]];
n = 4;
dom T = [0..n];
dom D = [1..n] × [1..(n + 1)];
dom D_1 = [1..n];
for (k : T) {
  if (k > 0) then
    apivot(k) = \max {|a(i, k, k − 1)| | k <= i <= n};
  if (k > 0) then
    ipivot(k) = \max {i | k <= i <= n : |a(i, k, k − 1)| = apivot(k)};
  if (k > 0) then
    forall (i : D_1)
      fac(i, k) = if (i <= k) then 1.0
                  || (i = ipivot(k)) then
                     a(k, k, k − 1)/a(ipivot(k), k, k − 1)
                  || else a(i, k, k − 1)/a(ipivot(k), k, k − 1)
                  fi;
  forall ((i, j) : D)
    a(i, j, k) = if (k = 0) then A_0[i, j]
                 || (i < k) then a(i, j, k − 1)
                 || (i = k) then a(ipivot(k), j, k − 1)
                 || (i = ipivot(k)) then
                    a(k, j, k − 1) − a(i, j, k − 1) ∗ fac(i, k)
                 || else a(i, j, k − 1) − a(ipivot(k), j, k − 1) ∗ fac(i, k)
                 fi;
}
```

Figure 3.6: The shared-memory version of the Gauss program.

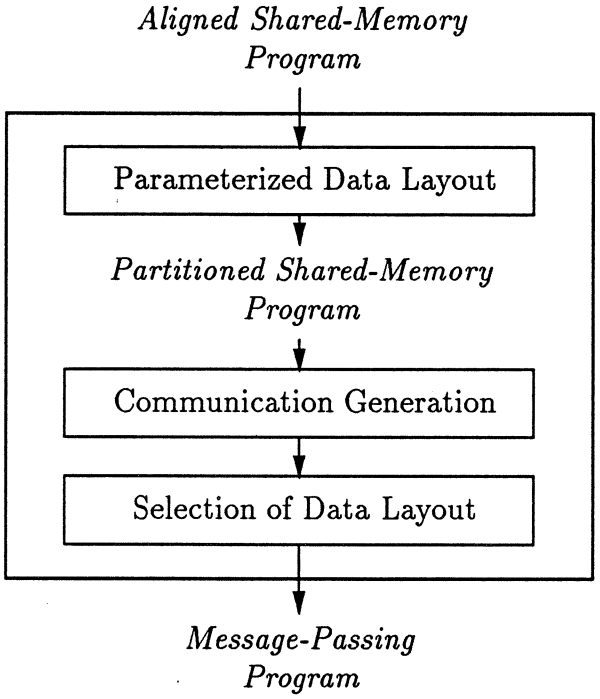


Figure 3.7: Generation of a message-passing program.

To resolve the cyclic dependence between data layout and communication generation, we break up the compilation process for transforming a shared-memory program into a message-passing program into several steps, as shown in Figure 3.7. In the parameterized data layout step, a set of parameters representing the most general data layout strategy is introduced to the aligned shared-memory program. The result can be represented in a partitioned shared-memory program. In the communication generation step, calls to communication routines are determined and inserted into the program. In the data-layout selection step, the cost of computation and communication is estimated according to the day layout parameters and a particular data layout strategy and parameters are selected. As the result, a message-passing program is generated. In the following, we describe each step in more detail.

3.2.1 Parameterized Data Layout

In Chapter 2, we described a set of data layout strategies for the compiler. All of those strategies can be viewed as degenerate cases of the most general one, the interleaved blocking strategy. To reduce the compilation cost, our compiler first generates the message-passing program with the most general strategy. It then sets the data layout parameters to appropriate values to generate programs with specific data layout strategies.

Using this approach, the compiler does not need to analyze data dependence and generate communications specifically for each data layout strategy. In addition, the user has the option of executing the compiled program experimentally to determine the best layout strategy and parameters if symbolic cost minimization is not used. It needs to be pointed out, however, that the generality of the target program does not come for free. Extra overhead may be incurred due to the bookkeeping process. It is therefore necessary for the compiler to perform some optimizations once a specific strategy is determined.

Example 3.6 Figure 3.8 shows two versions of a loop nest: the first one is just a shared-memory program block and the second is a version with parameterized data layout. In the second version, the original index domain D is partitioned into two parts: the processor domain E and the local index domain $\hat{D}$. Arrays with a hat, such as $\widehat{apivot}$, refer to the local segments of data fields in some specific processor; those without a hat, such as a, refer to data fields as a whole. References to a will be converted into references to $\hat{a}$ with communication at some later compilation stage.

3.2.2 Communication Generation

Given a parameterized data layout strategy, the compiler generates communication statements. The main issue is what type of communication statements to generate.

The compiler can quite straightforwardly place a pair of send-and-receive commands guarded by some predicates at each nonlocal data reference in the source

```
for (k : T) {
  forall ((i, j) : D)
    if ((k > 0) and (i = 1) and (j = k)) then
      apivot(k) = \max {|a(i, k, k − 1)| | k <= i <= n};
  };

forall ((p, q) : E)
  for (k : T) {
    forall ((î, ĵ) : D̂) {
      i = local_to_global(P, D, 1, î, (p, q));
      j = local_to_global(P, D, 2, ĵ, (p, q));
      if ((k > 0) and (i = 1) and (j = k)) then
        âpivot(k) = \max {|a(i, k, k − 1)| | k <= i <= n};
    };
```

Figure 3.8: A loop nest before and after parameterized data layout.

program. During the execution, the processor that needs the referenced data tests the predicate to determine whether it is available in the local memory of the same processor. If the data is not available locally, the send-and-receive statements are executed to bring in the data.

This naive approach ignores the reference patterns that may exist in the program, and may generate many pairs of unorchestrated messages and cause congestion in the network. Since both uniform and aggregate communication patterns occur extremely often (and sometimes exclusively) in scientific computation applications, it becomes worthwhile to identify these correlated communication patterns and invoke efficient routines to implement them.

The central idea of our communication analysis is to generate *efficient* communications for nonlocal data references. Our approach emphasizes the generation of collective communications, such as broadcasting, reduction, and array transpose. The

technique for generating communication statements consists of three steps:

(1) *Matching:* Match each reference pattern with the best communication routine.

(2) *Scheduling:* Insert the communication routines into the target program.

(3) *Synchronizing:* Set up appropriate guards for each communication routine.

The matching process works as follows. It first identifies the symbolic characteristic of the reference pattern, for instance, whether the data is to be moved from a single point in the domain or from multiple points. It then searches through the list of communication primitives for a matching one. The search is conducted in such a fashion that if there are multiple matching primitives, the most economical one (based on a communication metric) is encountered first and hence is selected. In the case where no matching primitive can be found, the reference pattern is decomposed into simpler subpatterns, and the matching continues on each of them recursively.

The scheduling process addresses the problem of placing communication commands in the appropriate location in the partitioned shared-memory program. Potential locations for communication commands are first identified, and the actual location is then selected. The decision-making problem involves trade-offs between communication cost, storage use, and other related factors.

The synchronization process ensures that the communication statements are properly synchronized.

Example 3.7 Figures 3.9 and 3.10 show the message-passing version of the Gaussian elimination algorithm. Compound communication statements in these figures are represented in an abstract form, $\langle \text{Comm} : \mathcal{P}, R \rangle$, where $\mathcal{P}$ represents a data layout strategy and R represents the communication routine being called.

3.2.3 Selection of Data Layout Strategy

After communication statements are generated with the most general data layout strategy, estimation of the costs of computation and communication is made and symbolic cost function is formulated. In the cases where program parameters and

```
dom T = [0..n];
dom D̂ = D̂(P, p);
dom E = E(P, D);
forall (p : E)
  for (k : T) {
    forall ((î, ĵ) : D̂) {
      (i, j) = Local_to_Global(P, D, (î, ĵ), p);
      if ((k > 0) and (i = 1) and (j = k)) then
        apivot(k) = \max {|a_1(i)| | k <= i <= n};
    };
    forall ((î, ĵ) : D̂) {
      (i, j) = Local_to_Global(P, D, (î, ĵ), p);
      if ((k > 0) and (i = 1) and (j = k)) then
        ipivot(k) = \max {i | k <= i <= n : |a_1(i)| = apivot(k)};
    };
    if (k > 0) then
      ⟨Comm: P, One-All-Broadcast(D, ipivot, ipivot_1)⟩;
    if (k > 0) then
      ⟨Comm: P, Spread(D, 1, k, a, a_3)⟩;
```

Figure 3.9: The message-passing version of the Gauss program (Part 1).

```
  forall ((î, ĵ) : D̂) {
    (i, j) = Local_to_Global(P, D, (î, ĵ), p);
    if ((k > 0) and (j = k)) then
      fac(î, k) = if (i <= k) then 1.0
                  || (i = ipivot_1) then a_2(k)/a_3(k)
                  || else â(î, k, k − 1)/a_3(k)
                  fi;
  };
  if (k > 0) then
    ⟨Comm: P, Spread(D, 2, k, fac, fac_1)⟩;
  forall ((î, ĵ) : D̂) {
    (i, j) = Local_to_Global(P, D, (î, ĵ), p);
    â(î, ĵ, k) = if (k = 0) then Â_0[î, ĵ]
                 || (i < k) then â(î, ĵ, k − 1)
                 || (i = k) then a_3(k)
                 || (i = ipivot_1) then a_2(j) − â(î, ĵ, k − 1) ∗ fac_1(i)
                 || else â(î, ĵ, k − 1) − a_3(j) ∗ fac_1(i)
                 fi;
  };
  ⟨Comm: P, Broadcast(D, 2, k, k, a, a_1)⟩;
  ⟨Comm: P, Spread(D, 1, k, a, a_2)⟩;
}
```

Figure 3.10: The message-passing version of the Gauss program (Part 2).

constants in the formula can be bound to values (for instance, through profiling the program), the cost function is evaluated to obtain the best data layout parameters. Once the best data layout strategy is chosen, communication statements can be further optimized, since some reference patterns no longer describe communication patterns across the processors.

Estimation of Communication Costs

When a specific target architecture is chosen, the cost of each communication routine can be estimated. For instance, suppose that the architecture is a hypercube and the index domain over which the routines are defined is embedded in the hypercube using some Gray coding. Then the complexity of a **One_All_Broadcast** will be $\mathcal{O}(M \log N)$ where N is the number of processors in the hypercube and M is the message size.

When a specific target machine is chosen, a more accurate cost function can be obtained. By running each communication routine on the actual machine, we can determine the constant factors in the complexity functions.

Given the estimations of communication routines, the communication cost of a program can be estimated as follows:

1. Compute the message size B_i for each communication routine C_i in the program. In general, B_i is function of the size of the program data structure S (e.g. size of an array) and the data layout parameters L. Since all the layout strategies under our consideration are special cases of the interleaved blocking strategy, B_i is formulated in terms of the most general case.

2. Estimate the cost of C_i, denoted f_i, based on the communication routine C_i and its message size B_i. The cost f_i is a function of data structure S, data layout parameters L, and the number of processors N.

3. Estimate the total communication cost of the program C by summing up $f_i(S, L, N)$ for all C_i according to the control structure of the program (i.e. number of iterations and conditionals).

If the data layout strategy is given by a set of parameters whose values are unknown at compile time, message sizes in communication routines therefore cannot be determined. However, symbolic expressions for the message sizes can still be obtained and the communication costs can be given in an unevaluated symbolic form. For simple cases, it is possible to compare expressions symbolically to determine the relationship between their values. In general, due to conditionals and unknowns, techniques such as profiling would be needed for cost estimation.

Communication Reduction

Communication reduction is an optimization step. When an index domain is not partitioned in every dimension, some interprocessor message-passing can be replaced by local memory accesses.

Example 3.8 Consider the spatial reference pattern $\ulcorner a@(i, j+2) \Rightarrow (i, j) : j > 0\urcorner$, which represents a communication pattern in a two-dimensional index domain. Suppose that a data layout strategy is chosen such that only the first dimension of the index domain is partitioned, then the communication pattern across the spatial index domain can be described by $\ulcorner a@(i) \Rightarrow (i) : j > 0\urcorner$, which represents a local memory access. Consequently, communication statements corresponding to the original reference pattern can be removed from the message-passing program.

3.3 Compilation of a Whole Program

The compilation techniques discussed in previous sections apply directly to program blocks rather than a whole program. In this section, we describe methods for linking program blocks to form a whole target program. We also describe a framework for balancing compiler optimizations at different levels.

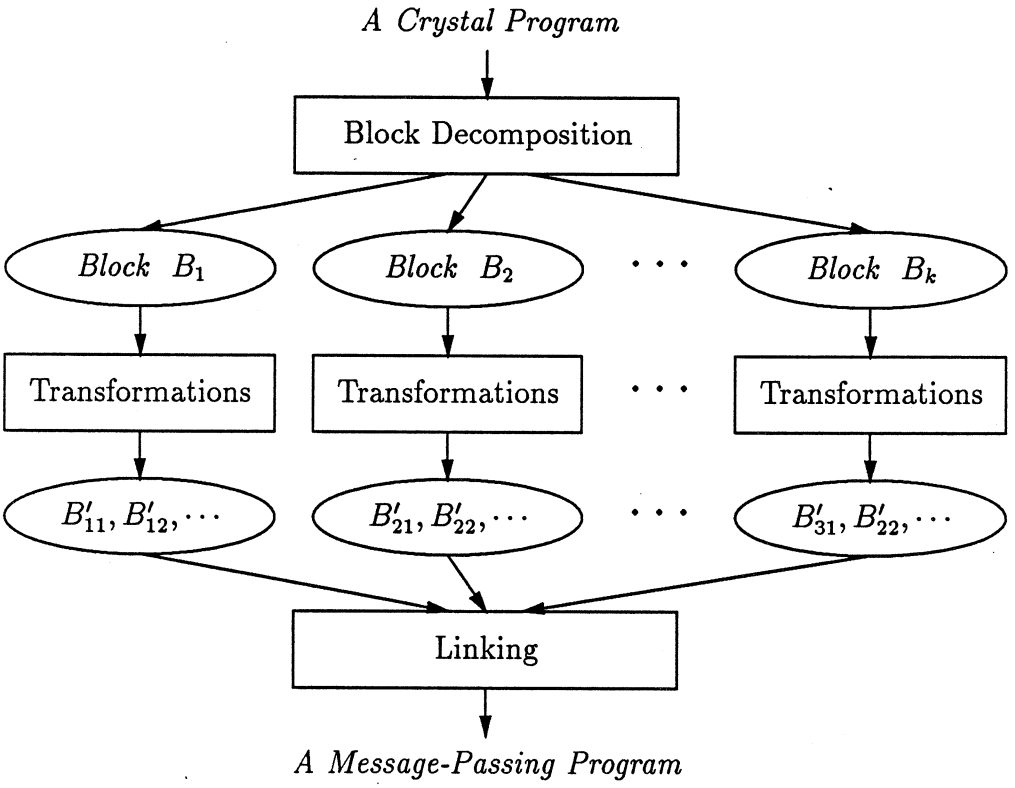


Figure 3.11: Compilation of a whole program.

3.3.1 Linking Program Blocks

The process of compiling a whole Crystal program works as follows. The program is first decomposed into program blocks (Section 3.1.2). Each block is then transformed through a sequence of intermediate forms: the aligned form, the shared-memory form, the partitioned form, and finally, the message-passing form. After these transformations, the individual program blocks are linked back together to form an integrated target program. Figure 3.11 illustrates this process. We present three block-linking strategies: *independent distribution*, *domain alignment*, and *block replication*. The following example is used to illustrate these strategies.

Example 3.9 Consider a program with two blocks:

```
for (y : D1)                    for (k : D3)
  forall (x : D2)                 forall ((i, j) : D4 × D5)
    a(x, y) = τ1;                   b(i, j, k) = τ2[a(i, k)];
```

In the first block, array a is defined over a two-dimensional domain $D_1 \times D_2$. In the second block, array b is defined over a three-dimensional domain $D_3 \times D_4 \times D_5$. Assume there is a data dependence from a to b.

Now suppose we have an abstract target machine with N processors. These processors can be configured either as a one-dimensional array $E_1 = [1..N]$ or as a two-dimensional array $E_2 = [1..N_1] \times [1..N_2]$ for some N_1 and N_2 that satisfy the relation $N_1 \times N_2 = N$.

Independent Distribution

In this strategy, program blocks are distributed over the processors of the target machine independently. Each block uses the best data layout strategy based on the analysis of intrablock communications. To guarantee interblock data dependences, communication statements are inserted between blocks to shuffle data from one distribution pattern to another.

Suppose that the independent distribution strategy is chosen to link the program blocks in Example 3.9. Then the first block will be distributed over a one-dimensional processor array E_1, since it matches the spatial index domain of the block; the second block will be distributed over a two-dimensional processor array E_2. Processors are fully utilized in both cases. However, during the execution of the program, the values of array a must be shuffled appropriately from a one-dimensional configuration to a two-dimensional configuration at the end to be used by b in the second block. Figure 3.12 (a) shows the resulting program and the arrangement of the processors in the target machine.

Independent distribution has the advantage that processors are fully utilized, since all the processors can participate in execution no matter which program block is in turn. However, interblock data shuffling may cause high overhead.

Domain Alignment

Domain alignment for linking blocks is very similar to domain alignment for aligning arrays within a block. The main difference is that for block linking, alignment is applied to the common domains of the blocks. It maps these domains to a common domain with the goal of minimizing the communication between blocks.

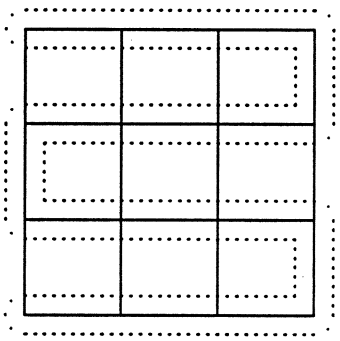
Suppose that the domain alignment strategy is chosen to link the blocks in Example 3.9. Based on the data dependences shown in the example, the spatial domain D_2 of the first block would be aligned to D_4, the first component of the spatial domain $D_4 \times D_5$ of the second block. The two blocks would be distributed to a two-dimensional processor array E_2. The communication between the two blocks is now in the form of simultaneous Spread along the second dimension of E_2, which is less costly than the data shuffling in the independent distribution case. However, during the execution of the first block, only some of the processors (i.e. those in the first column) are actively computing a, while others are sitting idle. The resulting program is shown in Figure 3.12 (b).

Domain alignment may reduce or eliminate interblock data shuffling. However, if some blocks have smaller or lower-dimensional domains than others, during their executions some processors may stay idle. Whether or not domain alignment is beneficial depends on the computation load of each block and the cost of interblock communication. The optimization problem is addressed later in this section.

Block Replication

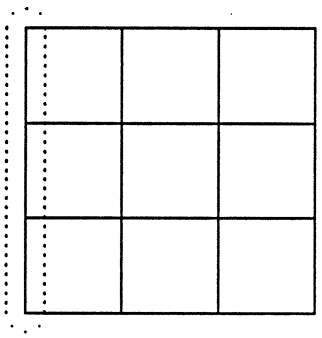
The block replication strategy applies to cases in which the index domains of program blocks are of different dimensionality. Consider two such blocks. In the replication

```
forall (p : E1)
  for (y : D1)
    forall (x : D̂2) a(x, y) = τ1;
⟨Comm: ⌜a(x) : E1 ⇒ (i, ∗) : E2⌝⟩;
forall ((q, r) : E2)
  for (k : D3)
    forall ((i, j) : D̂4 × D̂5)
      b(i, j, k) = τ2[a(i, k)];
```



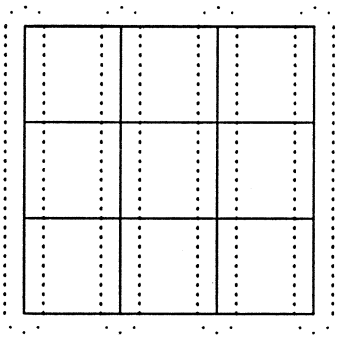
(a) Independent distribution.

```
forall ((q, r) : E2) {
  for (y : D1)
    forall ((i, j) : D̂4 × D̂5)
      if (j = 0) then a(i, j, y) = τ1;
  ⟨Comm: ⌜a(i, 0) ⇒ (i, ∗) : E2⌝⟩;
  for (k : D3)
    forall ((i, j) : D̂4 × D̂5)
      b(i, j, k) = τ2[a(i, 0, k)];
}
```



(b) Domain alignment.

```
forall ((q, r) : E2) {
  for (y : D1)
    forall ((i, j) : D̂4 × D̂5)
      a(i, y) = τ1;
  for (k : D3)
    forall ((i, j) : D̂4 × D̂5)
      b(i, j, k) = τ2[a(i, k)];
}
```



(c) Domain replication.

Figure 3.12: Three block-linking strategies.

strategy, the domain with lower dimensionality is replicated along some dimensions of the domain with higher dimensionality. The computation corresponding the replicated block is duplicated, but interblock communication may be reduced.

Suppose that the block replication strategy is chosen to link the blocks in Example 3.9. Then D_2 will be replicated along D_4, the first component of the spatial domain of b, and the interblock communication will be completely eliminated. Figure 3.12 (c) shows the result.

The block replication strategy can be used along with the domain alignment strategy: first apply the domain alignment strategy to select the best dimensions to align, then apply the block replication strategy to replicate the computation.

3.3.2 A Framework for Global Optimizations

Both the block-to-block transformations and the block linking process can produce many different outputs. To produce a single target program, selections must be made in both stages. We refer to selections occurring in the block-to-block transformations as *intrablock optimizations* and those occurring in the block linking stage *interblock optimizations*. Intrablock optimizations deal with the selection of multiloop nests and the selection of data layout strategies for a given program block. Interblock optimizations deal with the selection of block-linking strategies.

These two types of optimizations are closely related. On the one hand, intrablock optimizations reduce the number of program blocks that need to be handled at the interblock level, hence reducing the complexity of interblock optimizations. On the other hand, from the global point of view, it is desirable to keep more options open to interblock optimizations, since the compiler will then have more global information and hence will be able to make better selections. The following is the framework we use to resolve this dilemma:

1. Estimate both the computation and the communication costs for each program block. Based on the estimation, a weight factor is determined for each block.

The weight factor reflects the importance of the block to the whole program in terms of execution time.

2. Use local information to purge multiloop nests and/or data layout strategies that are known to be more costly (Section 3.2.3 and Section 5.3).

3. Use the weight factor information to further reduce the number of outputs from block-to-block transformations. More outputs are preserved for heavy-weighted blocks.

4. Compare the effects of different block-linking strategies on the available blocks, and select the best strategy.

The idea is to reduce the number of outputs to as low as possible when applying intrablock optimizations while avoiding purging options prematurely. For blocks that contribute a substantial amount to the total execution time of the program, more options should be kept open, since a small parameter change to these blocks can cause a big change to the overall performance. For blocks that do not consume too much computation resource, fewer options need to be considered.

Example 3.10 Consider a Crystal program with two program blocks. Suppose that the first block consumes 90% of the total computation and communication time of the program. Then selecting the control structure and data layout strategy for the second block becomes trivial, since the difference these choices can make has little effect on the global performance.

3.4 Code Refinement

The message-passing program generated by the compiler can be further transformed to more efficient forms by applying transformations similar to those used in restructuring Fortran programs (see, for instance, [PW86,Wol89]). This process is called *code*

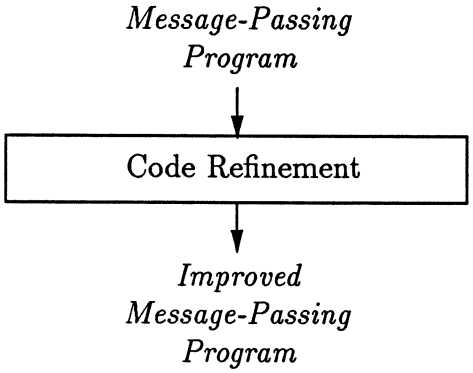


Figure 3.13: Code refinement.

refinement. Due to the language models we use, two transformations are particularly important: introduction of multiple assignments and elimination of common subexpressions. In the following, we discuss how these two optimizations affect performance of a program.

3.4.1 Introduction of Multiple Assignments

In our shared-memory program model any data object can be assigned to a value only once. This single-assignment property is preserved in domain partitioning and communication generation, hence it is also observed in the initial version of the message-passing programs. While it enables or simplifies compiler analysis and transformations, single-assignment may cause unnecessary memory consumption and computation overhead.

Example 3.11 Consider the loop nest in Figure 3.14 (a). Suppose that the only values of array a that need to be saved are those computed in the final iteration of the for loop. Then a two-dimensional array, as shown Figure 3.14 (b), would serve the need. In this new version, the result of the current iteration is stored in the same memory location as that of the previous iteration. It substantially reduces memory

```
for (k : T)
  forall ((i, j) : E)
    a(i, j, k) = τ[a(i, j, k − 1)];
```

(a)

```
for (k : T)
  forall ((i, j) : E)
    a(i, j) = τ[a(i, j)];
```

(b)

Figure 3.14: A simple example of introducing multiple assignment.

consumption.

In some cases, however, the reduction of memory consumption must be weighed against the increase of computation overhead, as shown in the following example.

Example 3.12 Consider the loop nest in Figure 3.15 (a). The memory allocated for the third dimension of array a cannot be completely eliminated, because values from two previous iterations must be accessible during the current iteration. Two alternatives are shown Figure 3.15 (b) and (c). In (b), two extra elements are used to hold previous values in each iteration and in (c), one extra element is used.

Both alternatives in the above example bring overhead to the loop nest. At some point, the overhead may overwhelm the benefit of storage saving. An evaluation according to some cost function should be made before applying the transformation.

Array assignment statements that are suitable for this transformation can be detected by the compiler. Array dimensions that correspond to for loops are candidates for elimination or reduction. Data dependence analysis with respect to for loop indices can reveal whether the corresponding array dimensions can be eliminated or reduced.

3.4.2 Common Subexpression Elimination

As the result of data layout, a nested forall loop may need to be executed on a single processor. The ordering of the loops in the nest, which does not affect the semantics

```
for (k : T)
  forall ((i, j) : E)
    a(i, j, k) = τ[a(i, j, k − 1), a(i, j, k − 2)];
```

(a)

```
for (k : T)
  forall ((i, j) : E) {
    a(i, j, 2) = τ[a(i, j, 1), a(i, j, 0)];
    a(i, j, 0) = a(i, j, 1);
    a(i, j, 1) = a(i, j, 2);
  }
```

(b)

```
for (k : T)
  forall ((i, j) : E)
    a(i, j, k mod 2) =
      τ[a(i, j, (k − 1) mod 2),
        a(i, j, (k − 2) mod 2)];
```

(c)

Figure 3.15: Another example of introducing multiple assignment.

of the loop nest, may have a big impact on the performance of the nest.

Example 3.13 Consider the loop nest in Figure 3.16 (a), which is to be executed on a single processor. Figure 3.16 (b) and (c) show two different loop orderings of the loop nest. In (b), the computation of array a does not depend on index j and hence is lifted out of the inner j loop, avoiding a great deal of redundant computation. In (c), such a lifting is not possible.

To determine whether a statement can be lifted from a loop, the compiler needs to analyze references of loop indices in expressions. Techniques for performing this optimization have become a standard component of conventional compilers.

```
forall ((i, j) : E1 × E2) {
  a(i) = τ1;
  b(i, j) = τ2;
}
```

(a)

```
forall (i : E1) {
  a(i) = τ1;
  forall (j : E2)
    b(i, j) = τ2;
}
```

(b)

```
forall (j : E2)
  forall (i : E1) {
    a(i) = τ1;
    b(i, j) = τ2;
  }
```

(c)

Figure 3.16: An example of common subexpression elimination.

Chapter 4

Index Domain Alignment

In this chapter, we discuss in detail the problem of index domain alignment. In Section 4.1, we illustrate the alignment problem with a group of examples and discuss the roles that domain alignment plays in the Crystal compiler. A reference metric for modeling data movement cost and a group of alignment functions are defined in Section 4.2. Section 4.3 presents a graph-theoretical formulation of the alignment problem. Practical alignment algorithms and a study of their performance are presented in Section 4.4. Finally, a proof of NP-completeness of the alignment problem is given in Section 4.5.

4.1 Domain Alignment and Its Roles in the Compiler

Index domain alignment refers to the process of aligning a group of index domains onto a single index domain. The alignment problem arises whenever multiple arrays are used in a program, which is the most common case for scientific subroutines and application programs.

4.1.1 Two Roles of Domain Alignment

In the Crystal compiler, index domain alignment plays two roles:

- It is a preprocessing step for the control structure synthesis module. It unifies dependences between different data fields into a single framework, enabling them to be represented by dependence vectors and direction vectors, which are the bases of the control structure synthesis algorithm.

- It is an optimization step for reducing data movement between arrays. Data fields in a Crystal program are eventually mapped to the distributed memory of the target machine. How these data fields are positioned with respect to each other has a direct impact on the performance of the program (to be elaborated below). Index domain alignment is the process that tries to find the optimal relative locations of data fields.

Reducing data movement is the major and more difficult role of index domain alignment. Such alignment is needed in any compiler system that compiles programs with arrays to distributed-memory machines. As for the role of a preprocessing step, as long as the given index domains are aligned to a single domain—whether the alignment reduces data movement or not—the task of unifying dependences is accomplished. Because of this, in the rest of this chapter, we focus our attention on the issue of reducing data movement.

It needs to be pointed out that with respect to the goal of reducing data movement, the index domain alignment module could have been placed after the control structure synthesis module. In that case, the alignment would be applied to the index domains of forall loops (i.e. the spatial index domains). To be consistent with the compiler structure shown in Figure 3.1 and to illustrate its double roles, we present index domain alignment with respect to Crystal programs. For a presentation based on the shared-memory program model, see [LC91b].

The following example illustrates the issue of reducing data movement.

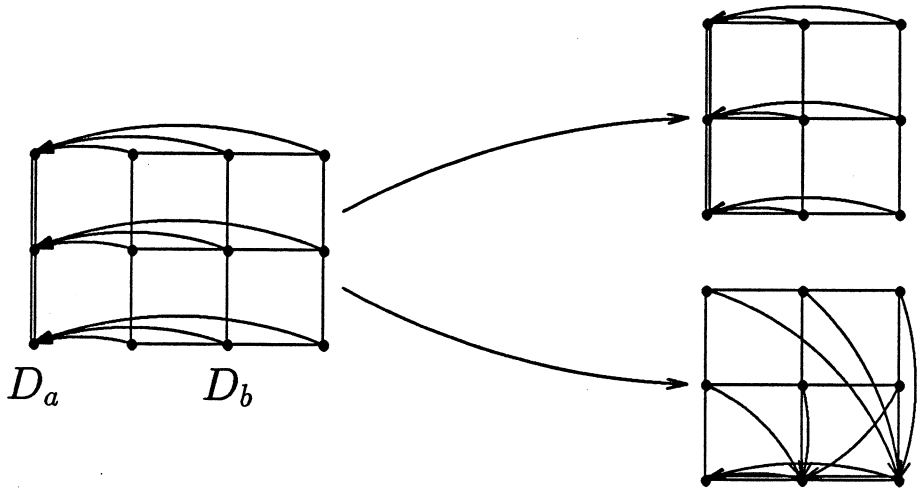


Figure 4.1: An example of index domain alignment.

Example 4.1 The following data field definition involves two data fields a and b:

$$\textsf{dfield}\ a(i) : [1..n] = \backslash + \{b(i,j) \mid 1 <= j <= n\}.$$

There are at least two possible ways to align these two data fields, as shown in Figure 4.1. The resulting data movement patterns in the common domain are different. Presumably one is better than the other, regardless of how the common domain is mapped to the distributed memory of the target machine.

Knobe et al. [KLS88,KLS90] developed a compiler technique for optimizing data allocation for the Connection Machine. In their method, the compiler does alignment automatically as part of the data allocation process. The usage patterns of array occurrences is analyzed to determine the allocation preferences of each occurrence. A simple zero-one metric—an allocation is either motion producing or motion free—is used to guide the alignment of each array occurrence.

4.1.2 Issues in Domain Alignment

To understand the alignment problem better, we present a group of examples below to illustrate the issues involved in domain alignment. In all of these examples, D denotes an index domain $[1..n] \times [1..n]$.

Example 4.2 Consider two data fields a and b defined over D.

$$\text{dfield } a(i,j) : D = b(j,i)$$
$$\text{dfield } b(i,j) : D = i + j$$

Even though the two arrays a and b are of exactly the same shape (defined by D), the distribution of their elements to processors need not necessarily be done in the same way. Due to the way b is referenced by a, it is beneficial to store $a(i,j)$ and $b(j,i)$ on the same processor to eliminate the need of communication. This simple example indicates that the compiler can choose the relative location of arrays by analyzing array reference patterns.

Example 4.3 dfield $a(i,j) : D = b(j,i) + b(i,j-2)$.

This example illustrates conflicting reference patterns. Whether $a(i,j)$ is aligned to $b(j,i)$ or $b(i,j-2)$, the data movement due to one of these two reference patterns cannot be reduced. Thus, the alignment problem must be formulated as an optimization problem where reference patterns may carry different weights.

Example 4.4 dfield $a(i,j) : D = \backslash + \{b(k,i) \mid 0 \leq k < n\}$.

This is a slightly more involved example where a reduction operator occurs in the dafinition of data field a. We consider first the simple scenario of mapping one element per processor. For each $(i,j) \in D$, a set of elements of b is referenced. If we store $a(i,j)$ and $b(i,j)$ at processor (i,j), then for each i we must do a reduction across the ith column of elements of b and distribute the result along the ith row. Alternatively,

if $b(j,i)$ is stored in processor (i,j), only a reduction operation over a row is needed, provided the processors are connected by a network such as a hypercube, a butterfly, etc., because the broadcast can be achieved at the same time as a side-effect of reduction [HJ88,JH87]. Thus alignment is related to the communication routines specific to the interconnection network of the target machine. The formulation of the alignment problem is therefore dependent on the communication cost. In the next section we will discuss the abstract model of a target machine and the metric we use to guide the alignment process.

Next, for the same example, suppose each domain is partitioned into subdomains, each of which is mapped to a processor: by aligning $a(i,j)$ with $b(j,i)$ for all $(i,j) \in D$ (i.e. transpose of b), and partitioning the domain along the first dimension (mapping a row into a processor), there will be no communication involved at all since the reduction operations now take place within a single processor. However, if $a(i,j)$ and $b(i,j)$ are mapped to the same processor, some communication must occur no matter how partitioning is done (by row or by column). This example illustrates that alignment always helps in reducing cross-references from a to b, independent of data partitioning across the processors.

Example 4.5 dfield $a(i,j) : D = b(j^2 - j, i + j)$.

This example illustrates an array reference containing a nonlinear expression. Though it might be possible to align a and b in such a way as to avoid communication, the cost of evaluating the extra conditional and nonlinear expressions generated by the alignment process may exceed the cost of communication. Thus, there are some tradeoffs involved in doing alignment. We will discuss the class of alignment functions under our consideration.

Finally, any compilation technique is limited by what is known at compile time, and alignment is no exception. Below is an example where an array reference contains an indirect reference $a(i, j-1)$ whose value may not be known until the program is in execution. Hence, such references shall not be taken into account by the alignment

algorithm.

Example 4.6 dfield $a(i,j) : D = b(a(i,j-1),\tau)$.

To summarize, alignment should use cost functions that reflect the communication costs on real machines and the symbolic forms of the alignment functions should be simple enough for a compiler to derive and to implement.

4.2 Reference Metric and Alignment Functions

In this section, we introduce concepts and notations for precisely describing the alignment problem.

4.2.1 Reference Metric

We classify reference patterns according to their *uniformity* (see Section 2.2): patterns that are uniform in every dimension; those that are nonuniform in one, two, three dimensions and so on; and those that are nonuniform in every dimension. In particular, we have the following special cases:

Local Memory Access: patterns that can be described by

$$\ulcorner a(i_1,\ldots,i_n) \leftarrow b(\tau_1,\ldots,\tau_n) : \gamma \urcorner \quad \text{where } \tau_1 \cong i_1,\ldots,\tau_n \cong i_n.$$

Neighborhood Access: patterns that are uniform in all the dimensions as in

$$\ulcorner a(i_1,\ldots,i_n) \leftarrow b(i_1+c_1,\ldots,i_n+c_n) : \gamma \urcorner \quad \text{where } c_1,\ldots,c_n \text{ are constants.}$$

Random Access: patterns that are nonuniform in all the dimensions.

The metric used for the alignment algorithm is built on this classification: the goal is to derive alignments that produce most-uniform patterns.

4.2.2 Alignment Functions

An index domain alignment can be thought of as a mapping between two index domains. An *alignment function* is a mapping from an index domain D to another index domain E. Correspondingly, a reference pattern will be transformed to a new form if the original domains of the two arrays are mapped to new ones. The goal of selecting alignment functions is to transform reference patterns to ones that have the least cost.

The uniformity notion defined earlier suggests that we relate the components of two index domains in such a way that maximum uniformity will be achieved. It also suggests that we reduce the constant offsets in each dimension of a reference pattern. Corresponding to these needs, we focus our attention on four simple types of alignment functions, namely, *permutation*, *embedding*, *shift*, and *reflection*. All these alignment functions have a simple symbolic form and are easy to compute. This is important because otherwise the transformed program may have a very high computation overhead. In addition, finding optimal alignment can be expensive.

Without loss of generality, we assume that all the index domains are aligned to a common domain, and all the components of the common domain are large enough to accommodate any component of the individual index domains. Each individual index domain is aligned to a subdomain of the common domain. The boundary information of each index domain is carried along when performing alignment, hence the corresponding subdomain can always be correctly defined. For simplicity, we omit the boundary conditions in the following definitions of alignment functions.

Definition 4.1 For two n-dimensional index domains, D and E, an alignment function $g : D \rightarrow E$ is said to be a *permutation* if

$$g(i_1, i_2, \ldots, i_n) = (i_{q_1}, i_{q_2}, \ldots, i_{q_n}) \tag{4.1}$$

where $(q_1, q_2, \ldots, q_n)$ is a permutation of $(1, 2, \ldots, n)$.

Definition 4.2 For an m-dimensional index domain D and an n-dimensional index domain E, where $m < n$, an alignment function $g : D \rightarrow E$ is said to be an *embedding* if

$$g(i_1, i_2, \ldots, i_m) = (i_{q_1}, i_{q_2}, \ldots, i_{q_n}) \tag{4.2}$$

where $(q_1, q_2, \ldots, q_n)$ is a permutation of $(1, 2, \ldots, n)$, and the expressions i_k, where $m < k \leq n$, are expressions that may contain $i_1, \ldots, i_m$.

For example, functions $g(i, j) = (i, 0, j)$ and $g(i, j) = (i, i+j, j)$ are both embeddings.

Definition 4.3 Let D and E be two interval domains. An index domain function $g : D \rightarrow E$ is said to be a *shift* if $g(i) = i - c$ where c is an integer.

Definition 4.4 Let $D = [l..u]$ and $E = [(-u)..(-l)]$. An index domain function $g : D \rightarrow E$ is said to be a *reflection* if $g(i) = -i$.

Permutation and embedding deal with transformation between different components of a domain (intercomponent), while shift and reflection deal with transformation within a given component (intra-component). The intercomponent alignment functions are useful in transforming reference patterns into more uniform ones. The intra-component alignment functions can be used to decrease the reference cost further by reducing the constant offsets.

Since the intercomponent and intra-component alignment functions are independent of each other, the domain alignment problem can be solved in two separate steps: first by considering permutation and embedding, and then shift and reflection. Retiming [LRS83] can be brought to bear on finding shifts. Our approach to finding reflections is ad hoc: we try to match special patterns. It turns out that generating appropriate permutations and embeddings is the central problem. Since it deals with only intercomponent alignment, we call this problem the *component alignment* problem, which is the focus of the following sections.

4.3 Modeling the Alignment Problem

4.3.1 Affinity of Domain Components

We first define a notion relating the components of two domains in a reference pattern.

Definition 4.5 Given a cross-reference pattern

$$\ulcorner a(i_1, \ldots, i_p, \ldots, i_m) \leftarrow b(\tau_1, \ldots, \tau_q, \ldots, \tau_n) : \gamma \urcorner$$

two domain components, $\mathsf{dom}(a, p)$ and $\mathsf{dom}(b, q)$, are said to have an *affinity* relation if $\tau_q \cong \ulcorner i_p + c \urcorner$, where c is a small constant.

The affinity relation between two domain components reflects a preference for aligning them, because aligning components according to their affinity relation will increase uniformity.

Example 4.7 From the reference pattern $\ulcorner a(i,j) \leftarrow b(j,i) : \gamma \urcorner$ two affinity relations can be derived, one between $\mathsf{dom}(a, 1)$ and $\mathsf{dom}(b, 2)$ and the other between $\mathsf{dom}(a, 2)$ and $\mathsf{dom}(b, 1)$. If the two domains are aligned according to these relations, $a(i, j)$ and $b(j, i)$ will be mapped to the same processor, and hence no communication is needed.

We want to point out that the definition of affinity depends on the definition of reference metric, which varies with the communication characteristics of the target machines and the degree to which one desires to model them. The affinity definition can always be refined to allow special patterns to be included.

4.3.2 Component Affinity Graph

We model component alignment as a graph problem. An undirected, weighted graph called a *component affinity graph* (CAG) is constructed from the source program based on the reference patterns as described below.

The nodes of the graph represent the components of index domains to be aligned. They are grouped in *columns*: each column contains those nodes representing components from the same index domain.

Using the concept of affinity, edges in a CAG are constructed as follows: For each distinct reference pattern (excluding self-reference) in the program, an edge is generated between two nodes if the two corresponding domain components have affinity. An edge is denoted by a pair $\langle \mathsf{dom}(a,i), \mathsf{dom}(b,j) \rangle$ where $\mathsf{dom}(a,i)$ and $\mathsf{dom}(b,j)$ are the two corresponding domain components.

Note that self-reference patterns are ignored, because alignment occurs only between index domains of different arrays. Also, different instances of the same reference pattern are considered only once since a datum can be referenced many times after it is communicated.

Using edges to represent affinity relations between domain components does not take into account the degree of alignment preference of each reference pattern. We introduce edge weights for this purpose.

Definition 4.6 Two or more edges in a CAG are said to be *competing* if they are generated by the same reference pattern and they are incident on the same node.

Example 4.8 Consider two reference patterns

$$\ulcorner a(i,j) \leftarrow b(i,i) : \gamma_1 \urcorner$$
$$\ulcorner b(i,j) \leftarrow a(i,j) : \gamma_2 \urcorner$$

From the first reference pattern, two competing edges, $\langle \mathsf{dom}(a,1), \mathsf{dom}(b,1) \rangle$ and $\langle \mathsf{dom}(a,1), \mathsf{dom}(b,2) \rangle$, are derived. From the second reference pattern, two noncompeting edges, $\langle \mathsf{dom}(a,1), \mathsf{dom}(b,1) \rangle$ and $\langle \mathsf{dom}(a,2), \mathsf{dom}(b,2) \rangle$, are derived. A noncompeting edge indicates a strong preference for aligning the two domain components. A competing edge indicates more than one equally good alignment in the absence of any noncompeting edge.

We assign weights to the edges of a CAG to reflect the strength of preference:

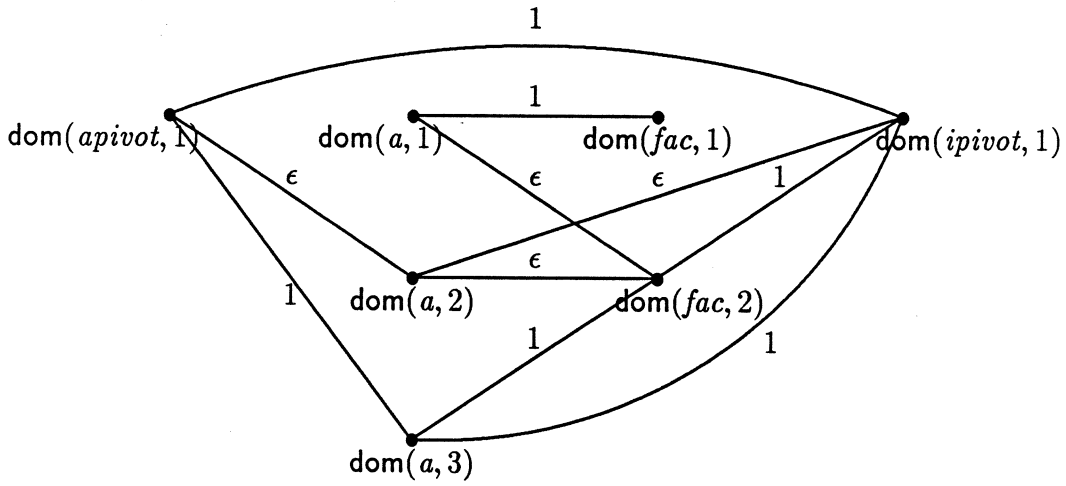


Figure 4.2: A component affinity graph.

- each noncompeting edge is assigned weight 1;
- each competing edge is assigned weight ϵ (a value much smaller than 1).

A CAG so defined may contain multiple edges between a pair of nodes since there might be multiple reference patterns between two arrays. The graph can be simplified by replacing each set of multiple edges with a single edge whose weight is the sum of their weights. Figure 4.2 illustrates the CAG of the Gaussian elimination program in Figure 3.6.

4.3.3 The Component Alignment Problem

Given a component affinity graph G as defined above, we can now define the component alignment problem as follows:

- *Legitimacy:* Let n be the maximum number of nodes in a column of G (i.e. n is the maximum dimensionality of all index domains to be aligned). Partition the

node set of G into n disjoint subsets $V_1, V_2, \ldots, V_n$, with the restriction that no two nodes belonging to the same column are allowed to be in the same subset.

- *Optimality:* The underlying idea is that those nodes in the same subset correspond to the domain components to be aligned. Since our goal of alignment is to align those components that have affinity, we want to partition the component affinity graph G so as to minimize the total weight of edges that are between those nodes that are in different subsets.

Among the set of index domains $\mathcal{D}$ to be aligned, choose one which is of the highest dimensionality n to be the *target domain* E. A permutation g mapping from a domain $D \in \mathcal{D}$ (also of dimensionality n) into the target domain E can be constructed using the above graph partition:

$$g : D \rightarrow E,\ g(i_1, i_2, \ldots, i_n) = (i_{q_1}, i_{q_2}, \ldots, i_{q_n})$$

where the nodes representing component D_i and component E_{q_i} are in the same subset.

Similarly, an embedding can be defined using the graph partition if D is of lower dimensionality than n. In this case, the components of the extra dimensions of E to which an element of D maps needs to be determined.

Example 4.9 Consider reference pattern $\ulcorner a(i) \leftarrow b(i, i+1) : \gamma \urcorner$. Suppose the only component of $\mathsf{dom}(a)$ is aligned to the first component of $\mathsf{dom}(b)$, we now need to decide where $\mathsf{dom}(a)$ should reside with respect to the second dimension in $\mathsf{dom}(b)$. We check the reference patterns and find that the second component of $\mathsf{dom}(b)$ is referenced once with an expression $i+1$ in the definition of a. Clearly, if this reference pattern is the only one between a and b we want to align $\mathsf{dom}(a)$ with those elements $(i, i+1)$ in $\mathsf{dom}(b)$.

In general, there can be more than one expression to be used if there are more reference patterns. Our approach is ad hoc in this case, using default constants such as the lower bounds of the interval domains.

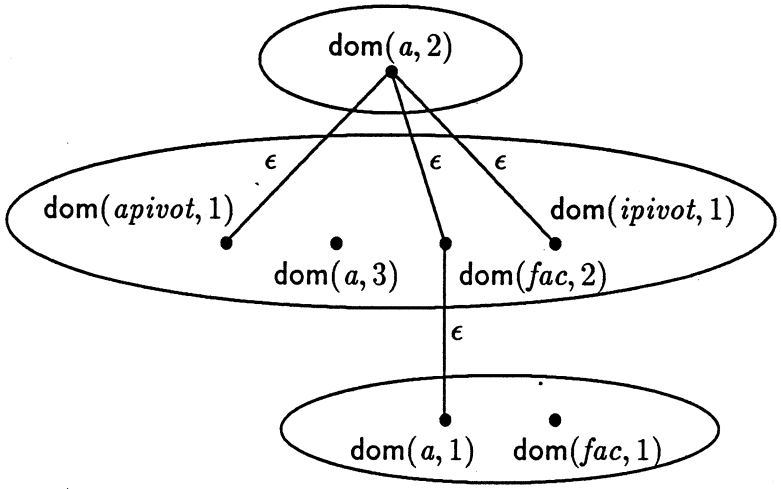


Figure 4.3: The optimal partition of the CAG.

4.3.4 Aligned Crystal Programs

Using the above definition, the optimal partition of the CAG of the Gaussian elimination program is illustrated in Figure 4.3.

The index domain of a, $\mathsf{dom}(a)$, is the target domain, since it is of highest dimensionality. $\mathsf{dom}(fac)$, which is two-dimensional, is embedded as a plane lying diagonally within $\mathsf{dom}(a)$ by aligning its components with the first and third components of $\mathsf{dom}(a)$. The domains of *ipivot* and *apivot* are embedded in the domain of fac at the same location. Formally, these alignment functions are defined as the following functions from the domain of an array to the target domain $\mathsf{dom}(a)$:

$$g_1 : \mathsf{dom}(fac) \rightarrow \mathsf{dom}(a),\ g_1(i,k) = (i,k,k)$$
$$g_2 : \mathsf{dom}(ipivot) \rightarrow \mathsf{dom}(a),\ g_2(k) = (0,k,k)$$
$$g_3 : \mathsf{dom}(apivot) \rightarrow \mathsf{dom}(a),\ g_3(k) = (0,k,k)$$

The alignment functions given above can now be used to transform the original Crystal program (Figure 3.4) into an *aligned* Crystal program as shown in Figure 3.5. This program transformation process is completely mechanical.

For this particular example, applying the optimal alignment results in a 20% reduction in communication cost compared to a straightforward default alignment.[1]

4.4 Practical Alignment Algorithms

The component alignment problem described above is, unfortunately, expensive to solve. A special case of the problem is one in which all the index domains are of two dimensions. We can reduce the simple max cut (MAXCUT) problem [GJ79] to this special case (with the number of index domains being the variable) and show it to be NP-complete.

Due to the NP-completeness result, we do not expect any polynomial algorithm to find the optimal alignment for index domains of dimensions higher than two. Unfortunately, a naive exhaustive search algorithm is not practical: for a group of six three-dimensional index domains, an exhaustive search algorithm may take two or more hours to find the optimal alignment on a Sun 3/50. Thus, we have devised the following practical alignment algorithms.

4.4.1 A Naive Algorithm

The naive algorithm is essentially a greedy algorithm where a single index domain is chosen at each step for aligning with the target domain, and there is no back-tracking. We use the fact that the problem of aligning two index domains is just a bipartite graph-matching problem and there exist efficient algorithms [Law76] to solve it. The naive algorithm is shown in Figure 4.4.

[1]The default alignment function for an m-dimensional index domain is to map its components to the first m components of the common domain with the original ordering.

Algorithm Naive_Alignment(G)

Comments: This algorithm runs in N steps, where N is the number of columns in a component alignment graph G. In each step, an arbitrary column is aligned to the target column by applying the optimal matching procedure to a bipartite graph constructed from the nodes in the two columns and the edges between them.

begin

$C_T \leftarrow$ *a column of* G *with the maximum number of nodes;*

$G_1 \leftarrow G$;

while G_1 *is not empty* **do**

Pick a new column, C_x;

$G_x \leftarrow$ Form_Bipartite_Graph(C_T, C_x, G_1);

$M \leftarrow$ Optimal_Alignment(G_x);

$G_1 \leftarrow$ Reduce_Graph(M, C_T, C_x, G_1);

end.

Procedure Form_Bipartite_Graph(C_T,C_x,G_1)

Comments: This procedure takes as input a CAG G_1 and two columns of nodes C_T and C_x, and returns a bipartite graph consisting only the two columns of nodes and the edges between them.

Procedure Optimal_Alignment(G)

Comments: This is for finding the optimal weighted matching for a bipartite graph G. See [Law76] for polynomial time algorithms.

Procedure Reduce_Graph(M,C_T,C_x,G_1)

Comments: Merge columns C_x and C_T by combining the matched nodes according to the matching M. "Clean" the graph by replacing multiple edges between two nodes with a single edge whose weight is the sum of their weights, and deleting all self cycles.

Figure 4.4: A naive index domain alignment algorithm.

Algorithm Heuristic_Alignment(G)

Comments: This algorithm has the same structure as the Naive alignment algorithm. The difference is in the procedure Form_Bipartite_Graph.

begin

$C_T \leftarrow$ *a column of* G *with maximum number of nodes;*

$G_1 \leftarrow G$;

while G_1 *is not empty* **do**

Pick a new column, C_x;

$G_x \leftarrow$ Form_Bipartite_Graph(C_T, C_x, G_1);

$M \leftarrow$ Optimal_Alignment(G_x);

$G_1 \leftarrow$ Reduce_Graph(M, C_T, C_x, G_1).

end.

Procedure Form_Bipartite_Graph(C_T,C_x,G_1)

Inputs: A CAG G_1 and two columns of nodes C_T and C_x.
Output: A bipartite graph with the transitive closure of the affinity relation.
Comments: Two nodes in the bipartite graph are connected by an edge if there is a path between these two nodes in G_1. The weight of such an edge is the sum of all the edge weights in the connected component of G_1 that contains these two nodes.

begin

for *each node pair* (x, y), *where* $x \in C_T$ *and* $y \in C_x$ **do**

$G_2 \leftarrow$ *the graph resulted from removing all the nodes in* C_T *and* C_x, *except for* x *and* y, *and all the edges that are incident on those nodes from* G_1;

Set up an edge between x *and* y *if they are connected in* G_2;

If the edge exists, assign its weight to be the sum of the weights of the connected component that x *and* y *belong to.*

end.

Figure 4.5: A heuristic index domain alignment algorithm.

4.4.2 A Heuristic Algorithm

The problem with the above naive greedy algorithm is that the quality of the alignment result is sensitive to the local alignment between two domains. For example, suppose we have two edges, $\langle \mathsf{dom}(a,2), \mathsf{dom}(b,1) \rangle$ and $\langle \mathsf{dom}(b,1), \mathsf{dom}(c,1) \rangle$, in a CAG. Even though there is no edge between $\mathsf{dom}(a,2)$ and $\mathsf{dom}(c,1)$, there is a preference to align these two nodes. What we really want is the transitive closure of the affinity relation. We therefore augment the bipartite graph obtained from CAG with a new edge between every pair of nodes that is connected in the original CAG. The weight of such an edge is defined precisely below. The result is a heuristic algorithm, shown in Figure 4.5.

4.4.3 Experimental Results

To see how the heuristic algorithm might work in general, we have conducted the following experiment: apply three different domain alignment algorithms to a large number of synthetic component alignment graphs. The three algorithms are: the *naive* greedy algorithm, the *heuristic* algorithm, and an *exhaustive-search* algorithm which produces the optimal result.

The data in each trial of the experiment is a synthetic component alignment graph. The edges in the graph are randomly generated according to a fixed density (i.e. an edge appears in the graph with the probability equal to the specified density). Edge weights are integers randomly chosen between 1 and 10, inclusive.

Fig 4.1 shows the results with graphs that consist of six columns with three nodes in each (i.e. six different arrays, each of dimensionality 3 or less). The density of the edges varies from 0.1 to 0.5. The middle three columns list the total edge weight of the resulting graph under each algorithm. The last two columns list the relative optimality of the results, which is defined to be the ratio of the total weight generated by an algorithm and the optimal weight.

Fig 4.2 shows the results of a set of experiments each of which assumes the number

Edge Density	*Num. Edges*	*Resulting Weights*			*Optimality*	
		Naive	*Heur*	*Exhaus*	N/E	H/E
0.1	17	54	31	22	2.45	1.41
0.1	18	61	43	30	2.03	1.43
0.1	14	51	23	15	3.40	1.53
0.1	10	28	10	10	2.80	1.00
0.1	11	39	33	29	1.34	1.14
Average		46.6	28	21.2	2.20	1.32
0.2	27	135	72	59	2.29	1.22
0.2	32	101	66	66	1.53	1.00
0.2	32	120	72	66	1.82	1.09
0.2	33	126	86	81	1.56	1.06
0.2	34	130	89	88	1.48	1.01
Average		122.4	77.0	72.0	1.74	1.08
0.3	39	154	110	103	1.50	1.07
0.3	43	129	98	98	1.32	1.00
0.3	39	150	136	104	1.44	1.31
0.3	55	203	160	153	1.33	1.05
0.3	44	154	120	113	1.36	1.06
Average		158.0	124.8	114.2	1.39	1.10
0.4	62	235	184	184	1.28	1.00
0.4	68	222	189	189	1.17	1.00
0.4	62	216	174	153	1.41	1.14
0.4	64	232	168	164	1.41	1.02
0.4	66	266	214	199	1.34	1.08
Average		234.2	185.8	177.8	1.32	1.04
0.5	66	292	213	206	1.42	1.03
0.5	71	301	248	229	1.31	1.08
0.5	79	347	232	232	1.50	1.00
0.5	70	255	216	199	1.28	1.09
0.5	77	350	295	288	1.22	1.02
Average		309	240.8	230.8	1.34	1.04

Table 4.1: Experimental results of three alignment algorithms (Case 1).

Dims	*Edge Density*	*Num. Edges*	*Resulting Weights*			*Optimality*	
			Naive	*Heur*	*Exhaus*	N/E	H/E
2	.3	8	39	18	18	2.17	1.00
	.3	13	44	35	35	1.26	1.00
	.3	7	15	6	6	2.50	1.00
	Average		32.7	19.7	19.7	1.98	1.00
2	.6	14	31	19	19	1.63	1.00
	.6	15	57	39	39	1.46	1.00
	.6	22	58	49	49	1.18	1.00
	Average		48.7	35.7	35.7	1.42	1.00
3	.3	18	57	50	42	1.36	1.19
	.3	20	62	47	44	1.41	1.07
	.3	15	51	36	36	1.42	1.0
	Average		56.7	44.3	40.7	1.39	1.09
3	.6	35	140	135	119	1.18	1.13
	.6	37	148	124	124	1.19	1.0
	.6	42	174	142	136	1.28	1.04
	Average		154.0	133.7	126.3	1.22	1.06
4	.3	37	165	131	122	1.35	1.07
	.3	35	182	126	117	1.56	1.08
	.3	39	180	137	136	1.32	1.01
	Average		175.7	131.3	125.0	1.41	1.05
4	.6	71	334	266	266	1.26	1.0
	.6	72	314	283	282	1.11	1.00
	.6	60	273	203	200	1.38	1.03
	Average		307.0	250.7	249.3	1.24	1.01
5	.3	61	261	208	—	—	—
	.3	50	239	174	—	—	—
	.3	52	210	187	—	—	—
5	.6	113	477	399	—	—	—
	.6	122	604	530	—	—	—
	.6	120	607	517	—	—	—
5	.3	78	404	299	—	—	—
	.3	75	365	282	—	—	—
	.3	77	377	284	—	—	—
6	.6	164	742	650	—	—	—
	.6	175	870	773	—	—	—
	.6	159	718	654	—	—	—

Table 4.2: Experimental results of three alignment algorithms (Case 2).

of columns (i.e., the number of arrays) is fixed at four while the highest dimensionality of the index domains varies from 2 to 6. The exhaustive algorithm takes too long to run when the highest dimensionality is greater than 5.

The naive algorithm generates alignments which are 22% – 120% more costly than the optimal ones. The alignments generated by the heuristic algorithm deviate from the optimal results by less than 10% in most of the cases. As to the times these algorithms take, both the naive and the heuristic algorithms run for periods of a few seconds to a few minutes, depending on the graph density, while the exhaustive algorithm runs for periods of 20 minutes to several hours or more.

4.5 NP-Completeness Result

The component alignment problem described above is, unfortunately, expensive to solve. A special case of the problem is one in which all the index domains are of two dimensions. We can reduce the simple max cut (MAXCUT) problem [GJ79] to this special case (with the number of index domains being the variable) and show it to be NP-complete.

We call an undirected graph G with an even number of nodes a *multipair* graph if the nodes are grouped in pairs and no edge exists between the two nodes of the same pair. A simple example of a multipair graph is the new graph obtained by putting together two copies of a given graph and pairing the nodes according to the isomorphic relation between the two copies.

Consider the domain alignment problem where all domains are two-dimensional. Its component alignment graph has the property that every column consists of exactly two nodes. Since no edge exists in any column of a CAG (due to the construction rule that self-reference patterns are ignored), a CAG is a multipair graph.

The problem of finding the optimal alignment can now be defined as follows:

Component Alignment Problem (ALIGN).
Instance: A multipair graph $G = (V, E)$ and a positive integer K.

Question: Can V be decomposed into two equal-size sets V_A and V_B by putting one node of each pair into V_A and the other into V_B, such that the number of edges bridging V_A and V_B is $< K$?

To show this problem to be NP-complete, the most natural NP-complete problem to use seems to be the minimum cut into bounded sets (MINCUT) problem. However, there is a little problem in getting the reduction to work. When we put together two copies of a general undirected graph (an instance of MINCUT) to form a multipair graph (an instance of ALIGN), the latter is already in the optimal form with respect to the ALIGN problem. To resolve this problem, we introduce the following problem.

Dual Component Alignment Problem (D-ALIGN).
Instance: A multipair graph $G = (V, E)$ and a positive integer K.
Question: Can V be decomposed into two equal-size sets V_A and V_B by putting one node of each pair into V_A and the other into V_B, such that the number of edges between V_A and V_B is $> K$?

Lemma 4.1 *The component alignment problem and its dual problem are equivalent.*

Proof:

This is simply because we can define a dual graph for each multipair graph G in which non-edges become edges and edges become non-edges. A mincut solution to G corresponds to a maxcut solution to the dual graph. □

Theorem 4.1 *The dual component alignment problem is NP-complete.*

Proof:

The problem is obviously in NP. To show it is NP-hard, we reduce an NP-complete problem, the *simple maxcut problem* (MAXCUT), to D-ALIGN. We show that for each instance of MAXCUT, we can construct an instance for D-ALIGN such that a solution to the latter can be transformed into a solution to the former.

First, we construct an instance. Let $G = (V, E)$ be an undirected graph serving as an instance of MAXCUT. Create an isomorphic copy of G and call it $G' = (V', E')$.

Let $\hat{G} = (V + V', E + E')$. Nodes in $V + V'$ are paired according to the isomorphic relation. $\hat{G}$ is a multipair graph and hence an instance of D-ALIGN.

Second, we convert a solution of D-ALIGN to that of MAXCUT. Suppose there is an algorithm solving D-ALIGN. Providing an input $\hat{G}$ to the algorithm, we get back a solution $(\hat{V_A}, \hat{V_B}, \hat{C})$, where $\hat{V_A}$ and $\hat{V_B}$ decompose the edge set of $\hat{G}$ and $\hat{C}$ is the set of edges which have one endpoint in $\hat{V_A}$ and one endpoint in $\hat{V_B}$.

The cut set $\hat{C}$ can be decomposed into two disjoint sets C and C', where C contains only edges from G and C' contains only edges from G'. Due to the isomorphic relation between G and G' and the rule of decomposition defined in D-ALIGN, we know that C and C' are isomorphic to each other. We claim that C is a maxcut (an edge set) of G. Assume it were not. Let C'' be a maxcut of G. Then $|C''| > |C|$. Let V_A'' and V_B'' be the decomposition of V corresponding to C''. For G', we have the isomorphic counterparts, C''', V_A''' and V_B'''. Now that $V_A'' + V_B'''$, $V_A''' + V_B''$ and $C'' + C'''$ define a solution to D-ALIGN. But $|C'' + C'''| > |C + C'|$. This contradicts the result that $\hat{V_A}$, $\hat{V_B}$ and $\hat{C}$ is a solution to D-ALIGN. □

Chapter 5

Control Structure Synthesis

In this chapter, we present an algorithm for synthesizing parallel and sequential control structures from a Crystal program and transforming the program into an explicitly parallel program. Section 5.1 gives an overview of our approach. Section 5.2 introduces some new notation for representing dependences of Crystal programs. The control structure synthesis algorithm is presented in Section 5.3. A proof of correctness of the algorithm is given in Section 5.4.

5.1 Overview of the Approach

In Crystal, parallelism is expressed through data fields and index domains. Because of the functional nature of a Crystal program, both parallel and sequential control information are implicit. We illustrate this with a simple example.

Example 5.1 Suppose we want to use the Jacobi iteration to solve the first-order finite difference equation corresponding to the Laplace's equation

$$4\Phi_{i,j} - \Phi_{i,j-1} - \Phi_{i,j+1} - \Phi_{i-1,j} - \Phi_{i+1,j} = 0.$$

In this method, the trial value $\Phi_{i,j}$ at the kth iteration is obtained by solving the kth equation for $\Phi_{i,j}$, using the values from the previous iteration at the four neighboring

```
dom D = [1..n] × [1..n]
dom T = [0..m]
dfield Φ(i, j, k) : D × T =
        if (k = 0) then a0(i, j)
        || (k > 0) then
            ω(Φ(i − 1, j, k − 1) + Φ(i, j − 1, k − 1)
             + Φ(i + 1, j, k − 1) + Φ(i, j + 1, k − 1))/4.0
        fi
```

Figure 5.1: Jacobi: A Crystal program for the Jacobi iteration.

nodes. A Crystal program can be easily written to implement this method and it is shown in Figure 5.1.

The Jacobi iteration method can be parallelized easily—the kth trial values at all nodes can be computed simultaneously. In other words, domain D represents a spatial domain and T represents a temporal domain. However, such information is not explicitly given by the Crystal program constructs. To the compiler, the two domains D and T bear no extra information except for their shapes and sizes. It is only through analysis that the compiler can find out that, for a fixed value of k, $\Phi(i, j, k)$ can be computed in parallel for all the elements (i, j) in domain D.

In order to generate parallel code for a parallel machine, it is necessary to make the parallelism in a program explicit. Our approach is to add control constructs such as for loops and forall loops to the program to explicitly represent sequential and parallel information. For the Jacobi iteration example, we can introduce a for loop over domain T and a forall over domain D. The resulting program is a shared-memory parallel program with explicit control information (Figure 5.2).

The Jacobi iteration example shows a straightforward transformation of a Crystal program to a shared-memory program: just adding two loop headings over the data field definition. In general however, the transformation is not always as simple. For

```
dom D = [1..n] × [1..n];
dom T = [0..m];
for (k : T)
  forall ((i, j) : D)
    Φ(i, j, k) = if (k = 0) then a0(i, j)
                 || (k > 0) then
                     ω(Φ(i − 1, j, k − 1) + Φ(i, j − 1, k − 1)
                      + Φ(i + 1, j, k − 1) + Φ(i, j + 1, k − 1))/4.0
                 fi;
```

Figure 5.2: The Jacobi program with explicit control information.

```
dom D = D1 × D2

dfield a(i, j) : D =
    a(i, j − 1) + b(i, j)

dfield b(i, j) : D =
    a(j − 1, j − 1) + 2
```

```
dom E = D2 × D1;
for (j : D2) {
  forall (i : D1)
    b(i, j) = a(j − 1, j − 1) + 2;
  forall (i : D1)
    a(i, j) = a(i, j − 1) + b(i, j);
}
```

Figure 5.3: The transformation of two dependent data fields to a multiloop nest.

instance, it is not always possible to transform two data fields into two independent loop nests. Mutually dependent data fields are likely to be transformed into a joint loop nest.

Example 5.2 Figure 5.3 shows the transformation of a Crystal program with two mutually dependent data fields. The transformed program consists of only one loop nest. Note that the iteration space of the loop nest, $E = D_2 \times D_1$, is a permutation of the original index domain $D = D_1 \times D_2$.

Our basic approach for synthesizing control structures for a Crystal program is to start with an aligned Crystal program, in which data fields belonging to a program

block (i.e. a group of mutually dependent data fields) are aligned to a common index domain. Using our approach, each program block is transformed into a multiloop nest. The transformation consists of two steps. In the first step, the common index domain of the program block is transformed into the iteration space of a multiloop nest. The domain of each loop in the nest is a component of the original index domain, and hence the iteration space of the multiloop nest is a permutation of the original index domain. In the second step, each data field definition is transformed into an array assignment statement. The changes to the program in this transformation are syntactic (e.g. removing the keyword **dfield**). Consequently, the transformed program preserves the structure and the single-assignment property of the Crystal program.

5.2 Dependence Representations

In Chapter 2, we defined reference patterns and the call-dependence graph to represent data dependences in a Crystal program. For convenience in presentation of the control structure synthesis algorithm, we define a few new notions of dependence in this section.

5.2.1 Dependence Vectors and Direction Vectors

Dependence vectors and direction vectors have been used to represent data dependences of sequential Fortran programs. Many vectorizing compilers use techniques based on analyzing these vectors (see for instance [Wol82]). We borrow these concepts to describe similar dependence information in a Crystal program.

Definition 5.1 Given a reference pattern $\ulcorner b(i_1,\ldots,i_n) \leftarrow a(\tau_1,\ldots,\tau_n) : D \mid \gamma \urcorner$, where data fields a and b are both defined over the same index domain D, the symbolic vector $(i_1 - \tau_1,\ldots,i_n - \tau_n)$ is called a *dependence vector.*

Definition 5.2 With respect to the dependence vector in Definition 5.1, the vector $(\mathsf{sign}(i_1 - \tau_1, D), \ldots, \mathsf{sign}(i_n - \tau_n, D))$ is called a *direction vector*, where

$$\mathsf{sign}(\tau[\boldsymbol{i}], D) = \begin{cases} - & \text{if } \tau[\boldsymbol{i}] < 0, \text{ for all } \boldsymbol{i} \in D; \\ = & \text{if } \tau[\boldsymbol{i}] = 0, \text{ for all } \boldsymbol{i} \in D; \\ + & \text{if } \tau[\boldsymbol{i}] > 0, \text{ for all } \boldsymbol{i} \in D; \\ * & \text{otherwise.} \end{cases} \tag{5.1}$$

A dependence vector simplifies the dependence information represented by a reference pattern. A direction vector further simplifies the dependence information represented by a dependence vector. Each element in a direction vector represents the collective information about the dependences of a reference pattern along a particular dimension of its index domain.

Example 5.3 Consider the following two reference patterns

$$\ulcorner b(i,j,t) \leftarrow a(3,j,t) : D \mid t > 1 \urcorner,$$
$$\ulcorner b(i,j,t) \leftarrow b(i,j,t-1) : D \mid t \leq 1 \urcorner.$$

The dependence vectors are $(i-3,0,0)$ and $(0,0,1)$, and the direction vectors are $(*,=,=)$ and $(=,=,+)$.

5.2.2 Dependence Tuples

In an environment in which data dependences between a group of data fields need to be analyzed in a single framework. The information represented by direction vectors alone is not specific enough. We introduce the following notion to be used as the major representation of data dependence in the rest of this chapter.

Definition 5.3 Given a reference pattern $\ulcorner b(i_1, \ldots, i_n) \leftarrow a(\tau_1, \ldots, \tau_n) : D \mid \gamma \urcorner$, the symbolic form

$$\ulcorner (d_1, \ldots, d_n)\ \langle b \leftarrow a \rangle : D \urcorner \tag{5.2}$$

is called a *dependence tuple*, where $(d_1, \ldots, d_n)$ is the direction vector.

Example 5.4 The dependence tuples for the reference patterns in Example 5.3 are $\ulcorner(*,=,=)\ \langle b \leftarrow a\rangle : D\urcorner$ and $\ulcorner(=,=,+)\ \langle b \leftarrow b\rangle : D\urcorner$.

A dependence tuple captures key aspects of the data dependences represented by a reference pattern: the names of the two data fields involved, the index domain, the direction of the dependences, and the direction vector. However, it glosses over some information that might be essential in certain cases, such as the conditionals in a reference pattern.

Definition 5.4 The data dependences represented by a dependence tuple as in Equation (5.2) are described by the following set

$$\{b(i_1,\ldots,i_n) \leftarrow a(i_1 - c_1,\ldots,i_n - c_n) \mid \\ c_k \in \mathsf{range}(d_k), 1 \le k \le n; (i_1,\ldots,i_n),(i_1 - c_1,\ldots,i_n - c_n) \in D\} \qquad (5.3)$$

where c_k's are integral variables and

$$\mathsf{range}(d_k) = \begin{cases} [-\infty..-1] & \text{if } d_k \text{ is } -; \\ [1..\infty] & \text{if } d_k \text{ is } +; \\ [0] & \text{if } d_k \text{ is } =; \\ [-\infty..\infty] & \text{if } d_k \text{ is } *. \end{cases}$$

Theorem 5.1 *To satisfy the data dependences represented by a reference pattern, it is sufficient to satisfy the data dependences represented by the corresponding dependence tuple.*

Proof:

Data dependences represented by a dependence tuple form a superset of the data dependences represented by the reference pattern from which the tuple is derived. □

dom $D = [1..n] \times [1..n]$
dfield $a(i,j) : D = b(i,j-1)$
dfield $b(i,j) : D = c(i,j) + c(i,j-1)$
dfield $c(i,j) : D = a(i,j) + 2$
dfield $d(i,j) : D = d(i+2,j-1)$

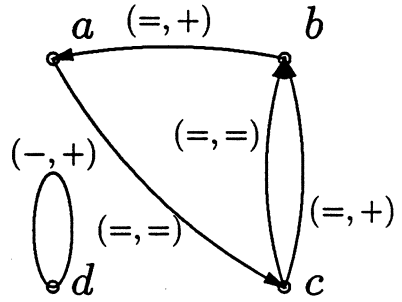


Figure 5.4: A set of aligned data fields and their augmented CDG.

5.2.3 The Augmented Call-Dependence Graph

The concept of a call-dependence graph (CDG) is defined in Chapter 3. In a CDG, each node represents a data field and each directed edge represents a call dependence. An augment call-dependence graph is an extension of a CDG in which auxiliary information is attached to the edges.

Definition 5.5 The augmented call-dependence graph for an aligned Crystal program is an acyclic graph. Each node of the graph corresponds to a data field in the program and each directed edge corresponds to a call dependence and is labeled by a direction vector. The direction of an edge indicates the call dependence.

Each edge in an augmented CDG can be described by a dependence tuple, and vice versa. Note that unlike the original CDG, the augmented CDG may have multiple edges between two nodes. Also note that the augmented CDG is defined for aligned Crystal programs, because the concepts of dependence vector and direction vector are not well-defined for unaligned Crystal programs.

Example 5.5 Figure 5.4 shows a group of aligned data fields and their augmented call-dependence graph.

5.2.4 Dependences with Respect to Domain Components

To determine what type of loop to introduce to a component of an index domain, we need to analyze data dependences with respect to domain components.

Definition 5.6 Consider a group of aligned data fields. Denote their common index domain by $D = D_1 \times \cdots \times D_n$, and the set of dependence tuples by T. With respect to T, the kth component of D (*i.e.* D_k) is said to be

(1) a *dependence-free component* if for every direction vector in T, its kth element d_k is $=$;

(2) a *dependence-carrying component* if for all the direction vectors in T,

(i) d_k is $+$ or $=$, and at least one d_k is $+$;

(ii) d_k is $-$ or $=$, and at least one d_k is $-$;

(3) a *dependence-conflicting component* if it does not belong to either of the above two cases.

For an index domain D, a component being dependence-free means that computation can be carried out in parallel with respect to that component; a component being dependence-carrying means that computation must be executed sequentially to preserve the dependences; and a component being dependence-conflicting means the dependences cannot be preserved through just ordering computation for that dimension—they have to be resolved through other components.

Example 5.6 With respect to the two dependence tuples in Example 5.4, $\ulcorner(*,=,=$ $)\ \langle b \leftarrow a\rangle : D\urcorner$ and $\ulcorner(=,=,+)\ \langle b \leftarrow b\rangle : D\urcorner$, the first component of domain D is a dependence-conflicting component, the second is a dependence-free component, and the third is a dependence-carrying component.

5.3 The Control Structure Synthesis Algorithm

In this section, we present an algorithm for transforming a group of aligned data fields defined over a common index domain into a multiloop nest of a shared-memory

program. The basic idea of the algorithm is to identify the ordering of the loops in the nest and to select an appropriate loop type for each loop.

Our algorithm bears a resemblance to Allen and Kennedy's algorithm [AK87] for parallelizing multiloop nests of Fortran programs. Both algorithms have similar input and output interfaces (e.g. labeled dependence graphs and multiloop nests) and use similar criteria for determining parallel loops (e.g. concepts like dependence-carrying and dependence-free).

The major difference of these two algorithms lies in the constraints associated with their inputs. In the case of Allen and Kennedy's algorithm, the input is a sequential Fortran loop nest. The algorithm tries to find vectorizable inner loops in the input loop nest, while preserving the loop nesting structure of the source program. Allen and Kennedy did point out that loop interchange techniques can be used to increase the flexibility of the algorithm's output.[1]

In our case, the input is a set of aligned data fields. There are no a priori loop nests. It is completely up to the compiler to determine the structure of the loop nest as well as the type of each individual loop. The framework of our algorithm therefore provides the maximum freedom for generating an optimal loop nest.

As a side note, it is interesting to know that our algorithm can also be applied to a dependence graph derived from a Fortran loop nest. However, the dependence-preserving property of our algorithm is too strong for such cases. Data dependences of a Fortran programs include three different types: *flow-dependences*, *anti-dependences* and *output-dependences* [Kuc78]. As pointed out by Allen and Kennedy in [AK87], only flow-dependences are the true dependences, because both anti-dependences and output-dependences can be eliminated or transformed to flow-dependences. An appropriate approach is to apply some transformations to eliminate or reduce anti- and output-dependences first and then apply our control structure synthesis algorithm.

[1] A recent study [Ban90] shows that using loop interchanges alone may not be sufficient to transform a loop nest into all the possible nesting structures.

Algorithm Synthesize_Control_Structure(D,G)

Comments: This is a recursive algorithm. The parameters D and G represent an index domain and an augmented CDG, respectively. Initially, D represents the common index domain of a set of aligned data fields and G their augmented CDG. The values of both D and G change during recursions.

begin

if G *is acyclic* **then**

  for *every element in* G *(denote* x*)* **do**

    $S_x \leftarrow$ Form_Forall_Loop*(*D, Form_AssignStat*(*x*))*;

  return Order_Sub_Structures(G,$\{S_x\}$);

else

  $X \leftarrow$ *the set of strongly connected components of* G;

  $\mathcal{G} \leftarrow$ *the (acyclic) graph of all the connected components;*

  for *every element in* X *(denote* G_k*)* **do**

    $V_k \leftarrow$ *the set of dependence vectors appearing as labels in* G_k;

    $C_k \leftarrow$ *the set of dependence-carrying components of* D *with respect to* V_k;

    if G_k *is a singleton*

    then $S_k \leftarrow$ Form_Forall_Loop*(*D, Form_AssignStat*(*G_k*))*;

    else if C_k *is empty* **then** $S_k \leftarrow$ Form_While-Active_Loop(D,G_k);

    else

      for *every component in* C_k *(denoted* D_i*)* **do**

        $D' \leftarrow$ *remove* D_i *from* D;

        $G' \leftarrow$ *remove from* G_k *all the edges whose corresponding dependences are carried by* D_i;

        $\hat{S}' \leftarrow$ Synthesize_Control_Structure*(*D',G'*)*;

        $S' \leftarrow$ Form_For_Loop(D_i,$\hat{S}'$);

      $S_k \leftarrow$ Select_Control_Structure($\{S'\}$);

  return Order_Sub_Structures($\mathcal{G}$,$\{S_k\}$);

end.

Figure 5.5: The control structure synthesis algorithm.

Procedure Form_AssignStat(x)

Inputs: x is a CDG node.
Output: An array assignment corresponding to the data field definition represented by x.

Procedure Form_Forall_Loop(D,S)

Inputs: S is a statement or a sub-control-structure returned from lower recursion stage and D is an index domain.
Output: If D is empty, then the result is just S; otherwise the result is a forall loop with D as the iteration space and S as the loop body.

Procedure Form_For_Loop(D_i,S)

Inputs: S is a statement or a sub-control-structure and D_i is a single domain component.
Output: A for loop with body S and index corresponding to i.

Procedure Form_While-Active_Loop(D,G)

Inputs: G is a call-dependence graph and D is an index domain.
Output: A while-active with D as the iteration space and data fields corresponding to the nodes in G as the loop body.

Procedure Order_Sub_Structures(G, $\mathcal{S}$)

Inputs: G is a call-dependence graph and $\mathcal{S}$ is a set of loop nests.
Output: A sorted sequence of loops in $\mathcal{S}$ according to dependences represented by G.

Procedure Select_Control_Structure($\mathcal{S}$)

Input: $\mathcal{S}$ is a set of loop nests.
Output: One loop from the set; the selection is made on a given criterion.
Comments: See Section 3.3 for more discussions on this procedure.

Figure 5.6: Subroutines of the synthesis algorithm.

5.3.1 The Algorithm

The control structure synthesis algorithm is shown in Figure 5.5. The subroutines used in the algorithm are shown in Figure 5.6.

The input to the algorithm is a group of aligned data fields. It is represented by two parameters: the common index domain and the augmented call-dependence graph. The algorithm generates a multiloop nest that preserves the data dependences of the data fields.

The control structure synthesis algorithm is a recursive algorithm. Its two parameters, D and G, are initially set to represent the common index domain and the augmented CDG of the input data fields, but their values are updated at every recursive stage. The algorithm works as follows.

Assume that the execution of the algorithm reaches the beginning of a recursive stage. Let D and G denote the values of the two parameters at the time. If G is acyclic, a statement is constructed for each node and a topological sorting is applied to G to arrange the statements into a right order that preserves the call dependences represented by G. The statement for a node is either an array assignment statement or a forall loop over an array assignment statement, depending on whether D is empty or not. The array assignment statements are constructed based on the data field definitions. The set of statements and loops constructed is returned as the result of the recursive stage.

If G is cyclic, the algorithm gets more complicated. First, strongly connected components of G are derived. Then the algorithm tries to construct a loop nest for each strongly connected component (to be described blow). If this step is successful for all the components, then a topological sorting is applied to the graph that is reduced from G by collapsing each strongly connected component into a node. The sorting result is used to arrange the statements and the for loops into the right order. Finally, the ordered list of statements and loop nests is returned as the result of the recursion stage. The case in which no for can resolve the dependence cycle in some

strongly connected components is discussed below.

A strongly connected component of G means that there is a dependence cycle among a group of data fields. We introduce a **for** loop to a domain component for each strongly connected component. The idea is that the **for** loop would carry some dependences, and hence it might break the dependence cycle. However, there are several cases: (1) more than one **for** loop (with different domain components) can break the dependence cycle; (2) a **for** loop reduces dependences, but not enough to break the dependence cycle; (3) no **for** loop can break the dependence cycle. Our method is to try every **for** loop that can reduce dependences. If there are more than one such loops, an evaluation is conducted after the loop nests are constructed. The loop which has the maximal parallelism is selected.[2] If after a **for** loop is constructed there are still dependence cycles in existence in the strongly connected component, the process has to be repeated. To do so, two new parameters D' and G' are created and the control structure synthesis algorithm is recursively applied upon them. The new domain D' is constructed from D by removing the component that has been chosen for the **for** loop. The new CDG G' is constructed from the strongly connected component by removing all the edges whose corresponding dependences are carried by the **for** loop.

The recursive process continues until it succeeds or until it exhausts the dependence-carrying components in D. The latter case means that there is still a dependence cycle and yet no **for** can be found to break it. For this case we enclose the data fields involved in the dependence cycle in a **While-Active** loop. The runtime system is responsible for executing a **While-Active** loop correctly and efficiently (see Section 2.4 for related discussions.)

[2] We also consider keeping several loop options open and make the evaluation and selection in the later compilation stage. Section 3.3 discusses this issue in more detail.

5.3.2 Selection of Multiloop Nests

The control structure synthesis algorithm can generate multiple outputs for a given input. Since the performance of a multiloop nest depends on many factors, and some of which are not available at the control structure synthesis stage, it is important not to throw away prematurely the control structures that may potentially have good performance. On the other hand, it is desirable to minimize the number of outputs so that the complexity of program optimizations can be reduced. We use the following strategy to purge the outputs of the synthesis algorithm as many as possible, while keeping those that are potentially usefully:

1. At each recursion stage, compare multiloop nests that are generated from the same input. Discard those that contain **while-active** loops, unless there is no other alternative.

2. At each recursion stage, compare multiloop nests that are generated from the same input. Discard those that have lower degrees of parallelism. The degree of parallelism can be measured by the number of **forall** loops contained in a loop nest.

5.3.3 An Illustrative Example

The control structure synthesis algorithm is illustrated here with a Crystal program. The program is for finding connected components in a undirected graph, based on an algorithm given in [Ull84]. The input to the algorithm is the adjacency matrix of the graph, and the output is an assignment of group numbers to the nodes in the graph. The algorithm works as follows. Initially each node is in a group by itself. Repeatedly, each group g finds the lowest-numbered group h to which it is adjacent, in the sense that there is an edge from some node in g to some node of h. If h is lower than g, g is merged into h, by giving the nodes of g the same number as h. After $\log n$ iterations, every group is a connected component by itself.

The original Crystal program and the aligned version of the program are shown in Figure 5.7 and Figure 5.8, respectively. In the original program, there are four data fields: $comp(v, i)$ represents the component of node v at stage i, which we take to be the smallest index of any node in that component; $min\_nbr(v, i)$ represents the minimum value of $comp(u, i)$ over all nodes u adjacent to v; $min\_comp(v, i)$ represents the minimum value of $comp(u, i)$ over all nodes u adjacent to component v at stage i, including vertex v itself; and finally, $next(v, i, \log n)$ represents the transitive closure of $min\_comp(v, i)$, i.e. the minimum value of $comp(u, i)$ over all components reachable from v at stage i; we know this is computable in at most $\log(n)$ steps, hence the value of the third index j.

To synthesize the control structure for the program, the augmented CDG of the aligned program is first derived. The CDG, as shown in Figure 5.9 (a), is cyclic. All the nodes in the graph belong to the same strongly connected component. According to the control structure synthesis algorithm, a for needs to be introduced to one of the three components of the common index domain D. With respect to the direction vectors, only the second component (corresponding to index i) is a dependence-carrying component. A for loop is thus chosen for that component (Figure 5.9 (b)). Three edges, labeled d_5, d_7, and d_8, are removed from G, since their dependences are carried by the component. The reduced CDG is shown in Figure 5.9 (c).

Next, find the strongly connected components of the new CDG and apply a topological sorting on them. As the result, three edges, labeled d_1, d_2 and d_6 respectively, are removed, and a partial loop nest is constructed (Figure 5.9 (d)).

There still exists a cyclic component. It consists of one node, labeled *next*, and two edges, labeled d_3 and d_4 (Figure 5.9 (e)). With respect to the two direction vectors, d_3 and d_4, the third domain component (corresponding to index j) is a dependence-carrying component. A for loop is thus constructed. The final loop nest is shown in Figure 5.9 (f).

The full-scale shared-memory program is shown in Figure 5.10. Note that in this program the domain components that are augmented during domain alignment

$n = 6$

$vset = \{v \mid 0 \leq v < n\}$

$E = \{[1,2],[1,3],[2,4],[2,1],[3,1],[4,2],[5,0],[0,5]\}$

dom $D_1 = [0..(n-1)]$

dom $D_2 = [0..\log(n)]$

dfield $comp(v,i) : D_1 \times D_2 =$
    if $(i = 0)$ then v
    || $(i > 0)$ then $next(v, i, \log(n))$
    fi

dfield $next(v,i,j) : D_1 \times D_2 \times D_2 =$
    if $(j = 0)$ then $min\_comp(v,i)$
    || $(j > 0)$ then $next(next(v,i,j-1),i,j-1)$
    fi

dfield $min\_comp(v,i) : D_1 \times D_2 =$
    $\min(comp(v,i-1),$
        $\backslash \min [min\_nbr(u,i) \mid u$ in $vset : comp(u,i-1) = v])$

dfield $min\_nbr(v,i) : D_1 \times D_2 =$
    $\backslash \min [comp(u,i-1) \mid u$ in $vset, [v,u]$ in $E]$

Figure 5.7: Ccomp: A Crystal program for finding connected components.

$n = 6$
$vset = \{v \mid 0 \leq v < n\}$
$E = \{[1,2],[1,3],[2,4],[2,1],[3,1],[4,2],[5,0],[0,5]\}$
dom $D = [0..(n-1)] \times [0..\log(n)] \times [0..\log(n)]$
dfield $comp(v,i,j) : D =$ ·
    if $(j = \log(n))$ then
      if $(i = 0)$ then v
      || $(i > 0)$ then $next(v,i,\log(n))$
      fi
    fi

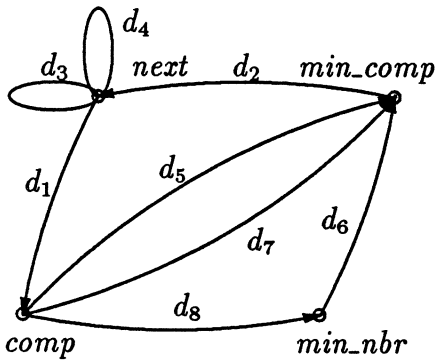
dfield $next(v,i,j) : D =$
    if $(j = 0)$ then $min\_comp(v,i,\log(n))$
    || $(j > 0)$ then $next(next(v,i,j-1),i,j-1)$
    fi

dfield $min\_comp(v,i,j) : D =$
    if $(j = \log(n))$ then
      $\min(comp(v,i-1,\log(n)),$
        $\backslash \min [min\_nbr(u,i,j)$
          $\mid u$ in $vset : comp(u,i-1,\log(n)) = v])$
    fi

dfield $min\_nbr(v,i,j) : D =$
    if $(j = \log(n))$ then
      $\backslash \min [comp(u,i-1,\log(n)) \mid u$ in $vset, [v,u]$ in $E]$
    fi

Figure 5.8: The aligned version of the Ccomp program.

d_1 : (=, =, *)
d_2 : (=, =, *)
d_3 : (=, =, >)
d_4 : (*, =, >)
d_5 : (=, >, *)
d_6 : (*, =, *)
d_7 : (*, >, *)
d_8 : (*, >, *)

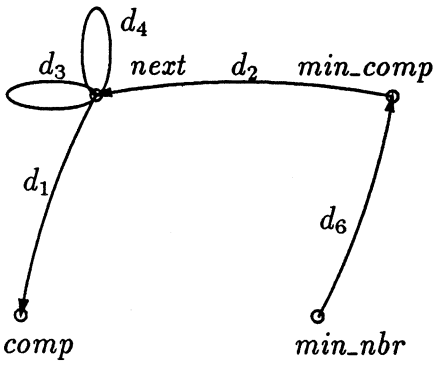


(a) Initial CDG.

```
for (i : D2)
  ⟨next, min_comp,
   comp, min_nbr⟩;
```

(b) A partial loop nest.

d_1 : (=, =, *)
d_2 : (=, =, *)
d_3 : (=, =, +)
d_4 : (*, =, +)
d_6 : (*, =, *)

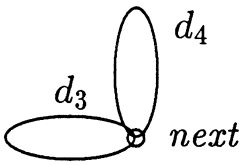


(c) After a for loop is introduced.

```
for (i : D2) {
  forall (v : D1)
    S(min_nbr);
  forall (v : D1)
    S(min_comp);
  ⟨next⟩;
  forall (v : D1)
    S(comp);
}
```

(d) Result of first iteration.

d_3 : (=, =, +)
d_4 : (*, =, +)



(e) Input to the second iteration.

```
for (i : D2) {
  forall (v : D1)
    S(min_nbr);
  forall (v : D1)
    S(min_comp);
  for (j : D3)
    forall (v : D1)
      S(next);
  forall (v : D1)
    S(comp);
}
```

(f) Final Result.

Figure 5.9: Synthesizing control structures for program Ccomp.

$n = 6;$
$vset = \{v \mid 0 \leq v < n\};$
$E = \{[1,2],[1,3],[2,4],[2,1],[3,1],[4,2],[5,0],[0,5]\};$
dom $D_1 = [0..(n-1)];$
dom $D_2 = [0..\log(n)];$

for $(i : D_2)$ {
  forall $(v : D_1)$
    $min\_nbr(v,i) = \backslash \min\,[comp(u,i-1) \mid u$ in $vset, [v,u]$ in $E];$
  forall $(v : D_1)$
    $min\_comp(v,i) = \min(comp(v,i-1),$
      $\backslash \min\,[min\_nbr(u,i) \mid u$ in $vset : comp(u,i-1) = v]);$
  for $(j : D_2)$
    forall $(v : D_1)$
      $next(v,i,j) =$ if $(j = 0)$ then $min\_comp(v,i)$
        || $(j > 0)$ then $next(next(v,i,j-1),i,j-1)$
        fi;
  forall $(v : D_1)$
    $comp(v,i) =$ if $(i = 0)$ then v
      || $(i > 0)$ then $next(v,i,\log(n))$
      fi;
}

Figure 5.10: The shared-memory version of the Ccomp program.

```
dom E_1 = D_{p_1} × ··· × D_{p_m};
dom E_2 = D_{p_{m+1}} × ··· × D_{p_n};
for ((i_{p_1}, ..., i_{p_m}) : E_1) {
  for/forall ((i_{p_{m+1}}, ..., i_{p_n}) : E_2)
    a(i_1, ..., i_n) = α;
  ...
  for/forall ((i_{p_{m+1}}, ..., i_{p_n}) : E_2)
    b(i_1, ..., i_n) = β;
}
```

Figure 5.11: The generic structure of a multiloop nest.

are removed. The arrays thus correspond to the data fields of the original Crystal program.

5.4 Validation Results

In this section, we present the conditions under which the data dependences of the original Crystal program are preserved in the transformed shared-memory program and we prove the correctness of the control structure synthesis algorithm. We first present some concepts and notations. Many of these concepts are developed from the ideas of [AK87].

5.4.1 Notation and Representations

Consider a set of aligned data fields:

$$\mathsf{dom}\ D = D_1 \times \cdots \times D_n$$

$$\mathsf{dfield}\ a(i_1, \ldots, i_n) : D = \alpha$$

$$\ldots$$

$$\textsf{dfield } b(i_1, \ldots, i_n) : D = \beta$$

The output of the control structure synthesis algorithm for these data fields is a multiloop nest, whose generic form is shown in Figure (5.11). According to the algorithm, the multiloop nest has the following features:

- The outer loops are for loops and the inner loops are forall loops. Furthermore, any loop containing two or more array assignment statements is a for loop. In other words, each forall loop encloses only one array assignment statement.
- Each array assignment statement corresponds to a data field definition and the correspondence is one-to-one.
- The loop iteration space $E_1 \times E_2$ is a permutation of index domain D, i.e. loop subscripts $(p_1, \ldots, p_n)$ is a permutation of array index subscripts $(1, \ldots, n)$.

For convenience, we introduce the following notations. The multiloop nest in Figure 5.11 is denoted by $\mathcal{L}$ and the individual loops in $\mathcal{L}$ are denoted by $L_1, L_2, \ldots, L_n$, according to their nesting level. The outermost loop is L_1 and the innermost is L_n. An array assignment statement in $\mathcal{L}$ is denoted by $S(a)$, where a is the array name. Specific instances of $S(a)$ are denoted using partial or complete indexing. Two indexing schemes are used: indices with parentheses referring to array dimensions and indices with square brackets referring to the nested loops. The following are a few specific examples:

- $S(a(i_1, \ldots, i_n))$ denotes the instance of $S(a)$ that assigns a value to the array element $a(i_1, \ldots, i_n)$.
- $S(a[j_1, \ldots, j_n])$ denotes the instance of $S(a)$ that appears in the loop iteration where $L_1, \ldots, L_n$ assume values $j_1, \ldots, j_n$, respectively.

The following relation holds for these two notations:

$$S(a(i_1, \ldots, i_n)) = S(a[i_{p_1}, \ldots, i_{p_n}]).$$

Whenever needed, we use $*$ to simplify the indexing expressions:

- $S(a(i_1, \ldots, i_m, *))$ denotes the set of instances of $S(a)$ that assign values to array elements $a(i_1, \ldots, i_m, i_{m+1}, \ldots, i_n)$ for all $(i_{m+1}, \ldots, i_n)$ ranging over $D_{m+1} \times \ldots \times D_n$.

- $S(a[j_1, \ldots, j_m, *])$ denotes the set of instances of $S(a)$ that appear in the loop iterations where loops $L_1, \ldots, L_m$ assume values $j_1, \ldots, j_m$, respectively.

- Both $S(a(*))$ and $S(a[*])$ denote the set of all instances of $S(a)$.

5.4.2 Data Dependences in a Multiloop Nest

Data dependences in a multiloop nest exist between array assignment statements. Informally, a statement $S(b)$ depends on a statement $S(a)$ if some instance of $S(b)$ uses the value created by some instance of $S(a)$.

Through the control structure synthesis, a data dependence from a data field element $a(\tau_1, \ldots, \tau_n)$ to a data field element $b(i_1, \ldots, i_n)$ becomes a data dependence from an array element $a(\tau_1, \ldots, \tau_n)$ to an array element $b(i_1, \ldots, i_n)$. Since multiloop nests obey the single-assignment rule, data dependence between two array elements can be expressed as dependence between two statement instances. Hence for the above example, the dependence from $a(\tau_1, \ldots, \tau_n)$ to $b(i_1, \ldots, i_n)$ becomes the dependence from $S(a(\tau_1, \ldots, \tau_n))$ to $S(b(i_1, \ldots, i_n))$.

Using the same idea, data dependences represented by a dependence tuple are transformed into data dependences between two sets of statement instances. The new form of dependences is described in the same dependence tuple notation, i.e. from

$$\boldsymbol{t} = \ulcorner(d_1, \ldots, d_n)\ \langle b \leftarrow a \rangle : D \urcorner \tag{5.4}$$

to

$$\boldsymbol{t} = \ulcorner(d_1, \ldots, d_n)\ \langle S(b) \leftarrow S(a) \rangle : D \urcorner. \tag{5.5}$$

For convenience, we also define a dual representation using loop subscripts:

$$\boldsymbol{t} = \ulcorner[d'_1, \ldots, d'_n]\ \langle S(b) \leftarrow S(a) \rangle : E \urcorner, \tag{5.6}$$

where $E = D_{p_1} \times \cdots \times D_{p_n}$ and $d'_i = d_{p_i}$, for $1 \leq i \leq n$.

Lemma 5.1 *Data dependences represented by a dependence tuple as in Equation (5.4) are preserved in the multiloop nest generated by the control structure synthesis algorithm, if the data dependences represented by the dependence tuple in Equation (5.6) are satisfied.*

Proof:

Follows from the previous discussions. □

The above lemma implies that the problem of preserving data dependences of a Crystal program is reduced to the problem of satisfying data dependences of the transformed shared-memory program.

Definition 5.7 Consider a multiloop nest $\mathcal{L}$ as in Figure 5.11 and a dependence tuple $\boldsymbol{t}$ as in Equation (5.6). A common loop of statements $S(a)$ and $S(b)$, L_k, is said to *carry* the dependences represented by $\boldsymbol{t}$ if

(1) d'_k is $+$ and L_k is an incremental for loop; or

(2) d'_k is $-$ and L_k is a decremental for loop.

The loop L_k is said to be *independent* of $\boldsymbol{t}$ if d'_k is $=$.

Definition 5.8 The common loops of statements $S(a)$ and $S(b)$ in $\mathcal{L}$ (i.e. the loops L_1 to L_m in Figure 5.11) are said to *carry* the dependences represented by $\boldsymbol{t}$ if there exists a k ($1 \leq k \leq m$) such that loops $L_1, \ldots, L_{k-1}$ are all independent of $\boldsymbol{t}$ and L_k carries the dependences of $\boldsymbol{t}$. The loop L_k is called the *leading dependence-carrying loop* in $\mathcal{L}$ for $\boldsymbol{t}$.

5.4.3 Conditions for Preserving Dependences

Data dependences in a shared-memory program are satisfied by executing statements in the right order. For example, in a multiloop nest, a data dependence from an instance of statement $S(a)$ to an instance of statement $S(b)$ is satisfied if the first instance is executed before the second one.

In the following presentation, a statement $S(a)$ is said to *precede* a statement $S(b)$ if $S(a)$ syntactically appears before $S(b)$ in the program.

Lemma 5.2 *If statement $S(a)$ precedes a statement $S(b)$ in their common loops $L_1, \ldots, L_m$, then for a fixed set of values, $i_1, \ldots, i_m$, any dependence from an instance in $S(a[i_1, \ldots, i_m, *])$ to an instance in $S(b[i_1, \ldots, i_m, *])$ is satisfied.*

Proof:

Follows the definition of the notation. □

Theorem 5.2 *The dependences represented by a dependence tuple $\boldsymbol{t}$ as in Equation (5.6) are preserved in a multiloop nest $\mathcal{L}$ if*

(1) $S(a)$ and $S(b)$ share no common loop and $S(a)$ precedes $S(b)$; or

(2) the common loops containing $S(a)$ and $S(b)$ carry the dependences of $\boldsymbol{t}$; or

(3) the common loops containing $S(a)$ and $S(b)$ are all independent of $\boldsymbol{t}$, and $S(a)$ precedes $S(b)$ in the common loops.

Proof:

In the first case, according to Lemma 5.2, any dependence from $S(a)$ to $S(b)$ is satisfied.

In the second case, assume that there are m common loops and the kth loop L_k ($1 \leq k \leq m$) is the leading dependence-carrying loop in $\mathcal{L}$ for $\boldsymbol{t}$. Then, according to the definition of the dependence-carrying loop, all the loops $L_1, \ldots, L_{k-1}$ are independent of $\boldsymbol{t}$. Hence the direction vector of $\boldsymbol{t}$, in the dual form, is $[=, \ldots, =, d'_k, \ldots, d'_n]$, where d'_k is either $+$ or $-$. According to Definition 5.4, the dependences represented by $\boldsymbol{t}$ are

$$\{b[j_1, \ldots, j_n] \leftarrow a[j_1, \ldots, j_{k-1}, j_k - c_k, \ldots, j_n - c_n] \mid$$
$$c_j \in \mathsf{range}(d'_j), k \leq j \leq n; [j_1, \ldots, j_n], [j_1, \ldots, j_{k-1}, j_k - c_k, \ldots, j_n - c_n] \in E\}$$

Assume that d'_k is $+$. Then $c_k \in [1..\infty]$ and L_k is an incremental for loop. Hence $S(a[j_1, \ldots, j_{k-1}, j_k - c_k, \ldots, j_n - c_n])$ are executed before $S(b[j_1, \ldots, j_n])$. All the

dependences represented by $\boldsymbol{t}$ are satisfied. The case in which d'_k is $-$ can be proven similarly.

In the third case, the direction vector of $\boldsymbol{t}$ is in the form $[=,\ldots,=,d'_{m+1},\ldots,d'_n]$, and the dependences represented by $\boldsymbol{t}$ are

$$\{b[j_1,\ldots,j_n] \leftarrow a[j_1,\ldots,j_m,j_{m+1}-c_{m+1},\ldots,j_n-c_n] \mid$$
$$c_j \in \mathsf{range}(d'_j), m < j \leq n; [j_1,\ldots,j_n],[j_1,\ldots,j_m,j_{m+1}-c_{m+1},\ldots,j_n-c_n] \in E\}$$

Since $S(a)$ precedes $S(b)$ in the common loops (L_1 through L_m), any dependence from $S(a[j_1,\ldots,j_m,*])$ to $S(b[j_1,\ldots,j_m,*])$ is satisfied (Lemma 5.2). Thus all the dependences represented by $\boldsymbol{t}$ are satisfied. □

5.4.4 Correctness of the Algorithm

Theorem 5.3 *A multiloop nest generated by the control structure synthesis algorithm preserves the dependences of the input data fields.*

Proof:

Let D and G be defined according to the algorithm. Let $\mathcal{L}$ denote a multiloop nest generated by the algorithm upon the inputs D and G. We prove the theorem by showing that for each dependence tuple $\boldsymbol{t}$, the data dependences represented by $\boldsymbol{t}$ are preserved in $\mathcal{L}$.

Denote the edge in G that corresponds to $\boldsymbol{t}$ by e. The following three cases cover all the possible situations that can occur for e during the execution of the algorithm (note that G changes during the execution of the algorithm):

1. e is removed from G due to the introduction of a for loop to a domain component of D and the dependences represented by $\boldsymbol{t}$ are carried by the loop;

2. e is removed from G during a reduction step, in which edges that do not belong to any strongly connected component of G are removed;

3. e remains in G until the end of the execution.

In the first case, assume the for loop is the kth loop in $\mathcal{L}$. Since e is not removed by loops $L_1, \ldots, L_{k-1}$, none of the loops carries the dependences of $\boldsymbol{t}$. According to the algorithm, no for loop is introduced to a dependence-conflicting domain component, thus $L_1, \ldots, L_{k-1}$ are all independent of $\boldsymbol{t}$ (Definition 5.7). This means the k loops $L_1, \ldots, L_k$ carry the dependences of $\boldsymbol{t}$ (Definition 5.8). According to Theorem 5.2, the dependences represented by $\boldsymbol{t}$ are satisfied in $\mathcal{L}$.

In the second case, assume the situation occurs in the body of the first k loops of $\mathcal{L}$. As in the first case, we can infer that loops $L_1, \ldots, L_k$ are all independent of $\boldsymbol{t}$. Furthermore, due to the topological sorting used in the algorithm, the subloop nest containing $S(a)$ precedes the subloop nest containing $S(b)$ in $L_1, \ldots, L_k$. According to Theorem 5.2, the dependences represented by $\boldsymbol{t}$ are satisfied in $\mathcal{L}$.

In the last case, according to the algorithm, a **while-active** loop is introduced to the component containing e. The dependences within component, including those represented by $\boldsymbol{t}$, are resolved at runtime. □

Chapter 6

Generation of Explicit Communication

In this chapter, we describe a compilation approach for automatically generating efficient communication from array references of a shared-memory parallel program. In Section 6.1, an overview of our approach is given. In Section 6.2, a set of communication routines defined with respect to an abstract machine model is introduced. The algorithm for matching reference patterns with communication routines and several related optimization issues are presented in Section 6.3. The methods for scheduling and synchronizing communication routines are described in Section 6.4 and Section 6.5, respectively. Section 6.6 contains discussions on alternatives and extensions to our approach.

6.1 The Communication Generation Problem

In compiling a shared-memory program to a distributed-memory target machine, explicit communication commands for interprocessor data transfer must be generated from the references in the source program.

The primary issue in the communication generation process is how to generate

efficient communication. It is fairly easy for a compiler to generate a naive send-and-receive pair to replace a nonlocal data reference in the source program, provided that data layout function is given. However, messages so generated are not globally orchestrated and may cause congestion in the network.

The other two important issues in the communication generation process are *scheduling* and *synchronization.* Scheduling refers to the process of placing communication commands in appropriate locations in the target program to ensure data dependences. Synchronization refers to the process of setting up correct conditions for invoking communication commands so that sends and receives in message-passing are correctly matched.

The approach we present in this chapter is based on matching program references with predefined communication routines. These routines implement various communication patterns using best algorithms on specific target machines. The compiler analyzes the syntactic reference pattern of each program reference and selects the most efficient communication routine for it. For the scheduling and synchronization problems, we present a solution which ensures that the data dependency of the shared-memory program is preserved in the message-passing program.

The problem of automatic generation of message-passing programs has been addressed by many other researchers [CK88,RS88,QHV88,ZBG88,KM89,RP89,KMSB90b]. In these systems, individual send and receive commands are automatically generated and inserted into the appropriate places in the target program. Optimizations are then applied to increase message granularity. Unique to our approach is the use of syntactic pattern matching to generate efficient communication.

6.2 Communication Routines

We select a set of communication patterns defined over an abstract machine and implement them as the *basic communication routines.* These routines are to be used in the target program to realize interprocessor communication. For a specific target

machine, when the embedding of the abstract machine is known, each communication routine can be carefully tuned for performance. This can be done by using a routing algorithm that takes advantage of the routine's particular communication pattern, or by special hardware support. For instance, for a hypercube target machine and a Gray-code embedding of the abstract machine, a one-to-all broadcast routine can be implemented with an $\mathcal{O}(\log(n))$ complexity.

There are two criteria for selecting communication patterns to be basic communication routines:

1. A basic routine must have an efficient implementation on the target machine.

2. The communication pattern of a basic routine must have some unique symbolic characteristics that the compiler can recognize.

There is no advantage in implementing a communication pattern as a basic routine if there is no special algorithm for it, for instance, in the case of the arbitrary pattern shown in Figure 2.2(d).

An efficient algorithm alone, however, is not enough. The compiler must be able to recognize symbolically the references that can exploit the efficient communication routines. Otherwise the communication routines must be explicitly called in the program, forcing the processor structure to be known. In addition, the data must be written or read via message buffers, as opposed to just being referenced in a shared address space.

How successful a compiler can be in recognizing patterns depends on the amount of explicit information available in the source program. For instance, with the use of explicit aggregate operators, identifying reduction or scan becomes trivial. However, different forms of permutations cannot always be identified.

Tables 6.1 and 6.2 show two lists of basic communication routines. Routines in Table 6.1 are called *general routines.* Those in Table 6.2 are called *simple routines.* For each general routine, the data movement crosses multiple dimensions of the index domain of the abstract machine. For each simple routine, the data movement is

confined to a single dimension of the index domain. There is a loose correspondence between the routines in the two tables, e.g., One-All-Broadcast corresponds to Spread, All-One-Reduce to Reduce, and so on. The cost column in each table shows a sample list of cost functions that are used to guide the compiler to match the best routine for a given reference pattern.

Tables 6.1 and 6.2 are by no means complete. More routines, such as gather, scatter, shuffle-exchange, and other special permutations can be included.

We now give a brief description of each of the routines in Table 6.1. In each routine, the parameter D is an index domain representing the abstract machine; a is a pointer to the input data; B denotes the size of the data; and a_1 is a pointer to the output data.

One-All-Broadcast$(D, \mathbf{s}, a, a_1, B)$: The data pointed to by buffer pointer a in virtual processor $\mathbf{s}$ in domain D is sent to all the other virtual processors in D. This pattern can be identified by the presence of a constant tuple in the source expression and indices in the destination expression.

All-One-Reduce$(D, \mathbf{d}, a, a_1, B, \oplus)$: A routine based on the reduction operator in APL. Data pointed to by a in every virtual processor in D are combined using the binary associative operator $\oplus$ and sent to virtual processor $\mathbf{d}$. This pattern is identified by the presence of a constant tuple in the destination expression and indices in the source expression.

Send-Receive$(D, \mathbf{s}, \mathbf{d}, a, a_1, B)$: Data pointed to by a in virtual processor $\mathbf{s}$ is sent to virtual processor $\mathbf{d}$. This pattern is identified by constant tuples appearing in both the source and destination expressions.

Uniform-Shift$(D, \mathbf{c}, a, a_1, B)$: Data pointed to by a is sent from every virtual processor $\mathbf{i}$ in D to virtual processor $\mathbf{i} + \mathbf{c}$. This pattern is identified by the presence of the same constant offset between the source and destination expressions over all virtual processors in D.

Routine	*Pattern*	*Cost*
One-All-Broadcast$(D, \mathbf{s}, a, a_1, B)$	$\ulcorner a@\ \mathbf{s} \Rightarrow \mathbf{i} : D \urcorner$	$\mathcal{O}(B \log N)$
All-One-Reduce$(D, \mathbf{d}, a, a_1, B, \oplus)$	$\ulcorner a@\ \mathbf{i} \Rightarrow \mathbf{d} : D \urcorner$	$\mathcal{O}(B \log N)$
Send-Receive$(D, \mathbf{s}, \mathbf{d}, a, a_1, B)$	$\ulcorner a@\ \mathbf{s} \Rightarrow \mathbf{d} : D \urcorner$	$\mathcal{O}(B)$
Uniform-Shift$(D, \mathbf{c}, a, a_1, B)$	$\ulcorner a@\ \mathbf{i} \Rightarrow \mathbf{i} + \mathbf{c} : D \urcorner$	$\mathcal{O}(B \log N)$
Affine-Transform$(D, M, \mathbf{c}, a, a_1, B)$	$\ulcorner a@\ \mathbf{i} \Rightarrow M\mathbf{i} + \mathbf{c} : D \urcorner$	$\mathcal{O}(B \log N)$

Table 6.1: General communication routines and their costs where B denotes the message size, and N the number of processors in D. Bold-face letters **i**, **s**, and **d** are shorthand for index tuples $(i_1, \ldots, i_n)$, $(s_1, \ldots, s_n)$, and $(d_1, \ldots, d_n)$.

Routine	*Pattern*	*Cost*
Spread(D, p, s, a, a_1, B)	$\ulcorner a@(l_1, s, l_2) \Rightarrow (l_1, i, l_2) : D \urcorner$	$\mathcal{O}(B \log N_p)$
Reduce$(D, p, d, a, a_1, B, \oplus)$	$\ulcorner a@(l_1, i, l_2) \Rightarrow (l_1, d, l_2) : D \urcorner$	$\mathcal{O}(B \log N_p)$
Multispread(D, p, a, a_1, B)	$\ulcorner a@(l_1, i, l_2) \Rightarrow (l_1, j, l_2) : D \urcorner$	$\mathcal{O}(BN_p)$
Copy(D, p, s, d, a, a_1, B)	$\ulcorner a@(l_1, s, l_2) \Rightarrow (l_1, d, l_2) : D \urcorner$	$\mathcal{O}(B)$
Shift(D, p, c, a, a_1, B)	$\ulcorner a@(l_1, i, l_2) \Rightarrow (l_1, i + c, l_2) : D \urcorner$	$\mathcal{O}(B \log N_p)$

Table 6.2: Simple communication routines and their costs where B denotes the message size, N_p the number of processors along the pth dimension of D, and l_1 and l_2 denote lists of indices $(i_1, \ldots, i_{p-1})$ and $(i_{p+1}, \ldots, i_n)$, respectively.

Affine-Transform$(D, M, \mathbf{c}, a, a_1, B)$: Input M is a constant $n \times n$ matrix, where n is the dimensionality of D. Data pointed to by a in every virtual processor $\mathbf{i}$ in D is sent to virtual processor $M\mathbf{i} + \mathbf{c}$. This pattern is identified by deriving both the sender's and the receiver's forms of the pattern, and verifying the relationship between the two forms. Note that Transpose is a special case of this routine.

Note that a composition of these routines can generate many more complex communication patterns. Thus a complex spatial reference pattern may be decomposed to match a composition of communication routines.

6.3 Matching Communication Routines

In this section, we describe an algorithm for matching the spatial reference patterns of a shared-memory program with the communication routines of an abstract machine. For a given program and a given data layout strategy, we define the abstract machine as being the same shape as the spatial index domain of the program. The matching algorithm is applied to one reference pattern at a time. Each reference pattern is matched with either a single routine or a composition of routines.

6.3.1 Definition of Matching

Definition 6.1 Given a syntactic reference pattern over an index domain D:

$$P : \ulcorner a@\sigma \Rightarrow \delta : D \mid \gamma \urcorner,$$

the set of *reference instances* represented by the pattern, denoted by $I_p(P)$, is defined as the set of pairs of elements of D obtained by replacing the index variables in the pattern by all possible values, disregarding the predicate[1]:

$$I_p(P) = \{(\sigma, \delta) \mid \sigma, \delta \in D\}.$$

Definition 6.2 Given a communication routine R with a syntactic communication pattern as described by P above, the set of *communication instances* represented by the routine, denoted by $I_r(R)$, is defined similarly:

$$I_r(R) = \{(\sigma, \delta) \mid \sigma, \delta \in D\}.$$

Example 6.1 Suppose that $D = [1..N_1] \times [1..N_2]$, then

$$I_p(\ulcorner a@(2,j) \Rightarrow (6,j) : D \mid j > 3 \urcorner) =$$
$$\{((2,1),(6,1)),((2,2),(6,2)),\ldots,((2,N_2),(6,N_2))\};$$

[1] Presently, our communication routines do not have predicates, hence the predicate in a reference pattern is not used in the matching.

$$
\begin{aligned}
&I_r(\mathsf{Spread}(D, 1, s, a, a_1, B)) = \\
&\quad \{((s,1),(1,1)),((s,1),(2,1)),\ldots,((s,1),(N_1,1)), \\
&\quad \cdots, \\
&\quad ((s,N_2),(1,N_2)),((s,N_2),(2,N_2)),\ldots,((s,N_2),(N_1,N_2))\}; \\
&I_r(\mathsf{Copy}(D, 1, s, d, a, a_1, B)) = \\
&\quad \{((s,1),(d,1)),((s,2),(d,2)),\ldots,((s,N_2),(d,N_2))\}.
\end{aligned}
$$

Definition 6.3 A communication routine R is said to *match* a reference pattern P if the set $I_r(R)$ for some choices of the routine's parameters is a superset of the set $I_p(P)$. A communication routine is said to be *perfectly matched* with a reference pattern if the two sets are exactly the same.

Example 6.2 Reference pattern $\ulcorner a@(2,j) \Rightarrow (6,j) : D \mid j > 3\urcorner$ can be matched with either $\mathsf{Spread}(D, 1, 2, a, a_1, B)$ or $\mathsf{Copy}(D, 1, 2, 6, a, a_1, B)$, but is matched perfectly only with the latter.

Forced Communication The communication routines discussed in Section 6.1 are all defined over regular index domains. By regular, we mean that a domain is either an interval domain or a Cartesian product of interval domains. But reference patterns of a shared-memory program may have associated predicates, which means that the required data movement may only occur in a selected part of a regular domain. For instance, the required data movement of the above reference pattern P is confined to a subdomain $[1..N_1] \times [4..N_2]$, specified by the predicate $j > 3$. As far as matching communication routines is concerned, such predicates are ignored, i.e., those processors that do not need data are nonetheless forced to participate in the communication and receive unneeded data. In the implementation, these extraneous data are discarded as soon as they arrive at the processor in order to free up the buffer space of the processor.

Algorithm Matching_Reference_Patterns(P)

Input: A reference pattern of a shared-memory program.
Output: A single communication routine or a composition of communication routines that best matches the reference pattern.

begin

Search through the list of routines from the simple ones to the general ones and try to find a perfect match for the given reference pattern;

For a simple pattern which fails the first step, find the lowest cost matching routine;

For a general pattern which fails the first step:

decompose it into subpatterns;

recursively apply the algorithm on the subpatterns;

optimize the composition of the resulting routines.

end.

Figure 6.1: A pattern-matching algorithm for generating communication routines.

6.3.2 The Matching Algorithm

The matching algorithm works as follows. Given a reference pattern, it first identifies the symbolic characteristic of the pattern, e.g. it decides whether the source expression is a constant tuple. It then searches through the list of communication routines for a match. The search is conducted in such a way that if there are multiple matching routines, the most economical one (based on the given cost of the routines) is encountered first and hence selected. In the case where no matching routine can be found, the algorithm breaks the reference pattern into simpler subpatterns, and works on each of them recursively. We first define the concept, then describe the algorithm and its major procedures.

Definition 6.4 A reference pattern whose source and destination tuples differ in only one element is called a *simple reference pattern*, such as:

$$\ulcorner a@(\sigma_1, \ldots, \sigma_{p-1}, \sigma_p, \sigma_{p+1}, \ldots, \sigma_n) \Rightarrow (\sigma_1, \ldots, \sigma_{p-1}, \delta_p, \sigma_{p+1}, \ldots, \sigma_n) : D \mid \gamma \urcorner$$

All other reference patterns are called *general patterns.*

Identifying a Perfect Match Identifying a perfect match is done by transforming the reference pattern into its *canonical form*[2] and symbolically comparing it with the data movement patterns of communication routines. We consider two cases.

Case 1: The input to the algorithm is a simple reference pattern as defined above.

Case 2: The input is a general reference pattern, as

$$\ulcorner a@\boldsymbol{\sigma} \Rightarrow \boldsymbol{\delta} : D \mid \gamma \urcorner$$

where $\boldsymbol{\sigma}$ and $\boldsymbol{\delta}$ are the source and destination expressions and γ is the predicate.

In Case 1, the following matching steps are taken:

Steps	*Comparisons*	*Matching Routines*
1.	**if** $\sigma_p \cong \delta_p$	**then** (Local Memory Access)
2.	**if** const$(\sigma_p) \wedge$ const(δ_p)	**then** Copy$(D, p, \sigma_p, \delta_p, a, a_1, B)$
3.	**if** const$(\delta_p - \sigma_p)$	**then** Shift$(D, p, \delta_p - \sigma_p, a, a_1, B)$
4.	**if** const(σ_p)	**then** Spread$(D, p, \sigma_p, a, a_1, B)$
5.	**if** const(δ_p)	**then** Reduce$(D, p, \delta_p, a, a_1, B, \oplus)$

The binary relation $\cong$ denotes two expressions having the same canonical form. The two predicates, const(σ_p) and formal(σ_p), test whether an expression σ_p contains constants only or indices only, respectively. Note that an expression containing temporal

[2] A canonical form of an expression is a syntactic form in which variables appear in a predefined order and constants are partially evaluated. For example, $\ulcorner 2-i+j \urcorner$ and $\ulcorner j-i+3-1 \urcorner$ would have the same canonical form $\ulcorner -i+j+2 \urcorner$. A canonical form of a reference pattern is a pattern in which all the expressions are in canonical forms. The process of deriving a canonical form is called *normalization*, and involves symbolic transformations and partial evaluations.

indices is considered a constant expression in the matching. The predicates in the left column are not mutually exclusive, so the order in which they are tested is important. If a pattern fails to satisfy any of the predicates in the above table, then it is not perfectly matched with a simple routine.

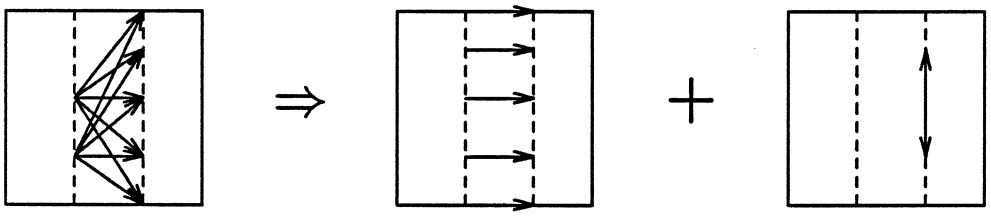
In Case 2, the following steps are taken:

Steps	*Comparisons*	*Matching Routines*
1.	**if** $\sigma \cong \delta$	**then** (Local Memory Access)
2.	**if** const(σ) $\wedge$ const(δ)	**then** Send-Receive($D, \sigma, \delta, a, a_1, B$)
3.	**if** const($\delta - \sigma$)	**then** Uniform-Shift($D, \delta - \sigma, a, a_1, B$)
4.	**if** const(σ) $\wedge$ formal(δ)	**then** One-All-Broadcast(D, σ, a, a_1, B)
5.	**if** formal(σ) $\wedge$ const(δ)	**then** All-One-Reduce($D, \delta, a, a_1, B, \oplus$)
6.	**if** affine(σ, δ)	**then** Affine-Transform($D, M, \mathbf{c}, a, a_1, B$)

The predicate affine(σ, δ) tests if the two vector expressions σ and δ have an affine relationship, i.e. if there exists a constant matrix M and a constant tuple $\mathbf{c}$, such that $\sigma = M\delta + \mathbf{c}$. If a pattern fails to satisfy any of the predicates in the above table, then it is not perfectly matched with a general routine.

Matching a Simple Pattern A simple reference pattern is always matched with a single routine. If a perfect match cannot be found, a less-perfect one is selected. In the worst case, the routine Multispread(D, p, a, a_1, B) is selected. It can match any simple reference pattern.

Decomposing a Reference Pattern A general reference pattern over an n-dimensional index domain can be thought of as a composition of n simple patterns, each describing data movement along one dimension. We call a composition of a subset of these simple patterns a *subpattern* of the general pattern. When a general reference pattern cannot be perfectly matched with a single routine, it is decomposed into subpatterns. The matching algorithm is then applied to these subpatterns recursively.



$\ulcorner a@(c(i,j), j-3) \Rightarrow (i,j) \urcorner$ Shift$(D,2,3,a)$ Multi-Spread$(D,1,a)$

Figure 6.2: Decomposition of a reference pattern.

Example 6.3 Reference pattern $\ulcorner a@(c(i,j), j-3) \Rightarrow (i,j) : D \urcorner$ cannot be perfectly matched with a single routine. It is therefore decomposed into two subpatterns, each of which is matched with a single routine:

$$\ulcorner a@(c(i,j), j-3) \Rightarrow (i, j-3) : D \urcorner \quad \textsf{Multispread}(D,1,a,a_1,B),$$
$$\ulcorner a@(i, j-3) \Rightarrow (i,j) : D \urcorner \quad \textsf{Shift}(D,2,3,a,a_1,B).$$

The composition of Shift$(D,2,3,a,a_1,B)$ and Multispread$(D,1,a,a_1,B)$ is the result of the matching algorithm (Fig 6.2).

In general, when the index domain is of high dimensionality, there are many ways to decompose a reference pattern. To find the optimal composition of routines, dynamic programming techniques are used.

Optimizing the Composition of Routines Notice that the ordering of the routines in the composition does not affect the correctness of the target program. However, it does affect the cost of communication. For the above example, we can have two orderings:

Case 1: Shift$(D,2,3,a,a_1,B) \circ$ Multispread$(D,1,a,a_1,B)$
(multispread in dimension one followed by shift in dimension two)—The message size for multispread is the original message size B, and the cost of it is $\mathcal{O}(B|D_1|)$. However, the result of multispread is that every processor gets a whole column

of data, hence the data size for shift becomes $B|D_1|$. The corresponding cost is $\mathcal{O}(B|D_1|)$.

Case 2: Multispread$(D, 1, a, a_1, B) \circ$ Shift$(D, 2, 3, a, a_1, B)$
(shift in dimension two followed by multispread in dimension one)—The message size for shift is B, so the cost is $\mathcal{O}(B)$. The message size for multispread is the same, and the cost is $\mathcal{O}(B|D_1|)$. The total cost is less than that of the first case.

The principle for ordering routines in a composition is to have them appear in the following order:

1. *Message-reducing* routines, such as Reduce and All-One-Reduce;
2. *Message-preserving* routines, such as Copy, Shift, Uniform-Shift, Send-Receive, and Affine-Transform;
3. *Message-broadcasting* routines, such as Spread, Multispread, and One-All-Broadcast.

6.3.3 Some Optimizations

Reference patterns derived directly from the input program are usually not in the most efficient form. In the following, we introduce several optimizations which transform the original set of reference patterns into ones better suited for implementation.

Constant Propagation

By constant propagation, the communication routine that perfectly matches a given reference pattern may change to another with lower cost. For example, a perfect match of the reference pattern

$$\ulcorner a@(3, j) \Rightarrow (i, j) : D \mid i = 4 \urcorner$$

would result in a Spread. However, the information that i is a constant or is space-invariant results in the reference pattern

$$\ulcorner a@(3,j) \Rightarrow (4,j) : D \urcorner,$$

which matches perfectly with a Copy, which is less costly than a Spread.

Combining Identical Patterns

Two reference patterns can be combined if they are equivalent, disregarding the guards. Clearly, the advantage of combining is that it reduces the number of communication statements and thus the number of messages. For example, the following two reference patterns

$$\ulcorner a@(i+3,j) \Rightarrow (i,j) : D \mid i > 53 \urcorner,$$
$$\ulcorner a@(i+3,j) \Rightarrow (i,j) : D \mid i < 7 \urcorner$$

can be combined as

$$\ulcorner a@(i+3,j) \Rightarrow (i,j) : D \mid i > 53 \text{ or } i < 7 \urcorner,$$

thus eliminating a Shift.

Combining Subset Patterns

If the set of instantiated source-destination pairs of one reference pattern is a subset of another, then it can be eliminated. For example, the following two reference patterns

$$\ulcorner a@(2,3) \Rightarrow (2,j) : D \mid j > 1 \urcorner,$$
$$\ulcorner a@(2,3) \Rightarrow (i,j) : D \mid j > 1 \urcorner$$

can be combined into

$$\ulcorner a@(2,3) \Rightarrow (i,j) : D \mid j > 1 \urcorner.$$

Aggregating Patterns

In the case that there are many individual reference patterns sending messages from the same source to different destinations, it is often better to use Spread instead of many Copys. For example, consider the following reference patterns

$$\ulcorner a@(2,j) \Rightarrow (c_1,j) : D \mid j > 1 \urcorner,$$
$$\ulcorner a@(2,j) \Rightarrow (c_2,j) : D \mid j > 1 \urcorner,$$
$$\cdots$$
$$\ulcorner a@(2,j) \Rightarrow (c_k,j) : D \mid j > 1 \urcorner$$

where $c_1, c_2, \ldots, c_k$ are constants. When k is large, it is better to combine these reference patterns as

$$\ulcorner a@(2,j) \Rightarrow (i,j) : D \mid j > 1 \urcorner.$$

When to do this optimization depends on the relative costs of Copy and Spread and must be determined experimentally for each target machine.

6.4 Scheduling Communication Routines

In this section, we consider the problem of inserting a communication routine into the target program. We first introduce some concepts and notation, then present two simple scheduling strategies.

6.4.1 Concepts and Notation

In Section 2.6, the concepts of computation segment and communication segment are defined for a message-passing program. Loosely speaking, with respect to a multi-loop nest (Figure 2.8), a computation segment is an inner loop nest consisting an array assignment statement, and a communication segment is a group of statements involving in a message-passing. We introduce some notation to represent these two concepts.

Assume $\mathcal{L}$ is a general loop nest over an n-dimensional index domain $D \times T$, and its first k loops are **for** loops over domain T. Let $L_1, \ldots, L_n$ denote the n levels of loops.

We denote a computation segment that consists of loops $L_{k+1}, \ldots, L_n$ and an assignment statement of array a by $\mathsf{S}(a, D, i_1, \ldots, i_k)$. The variables $i_1, \ldots, i_k$ are the indices of the first k **for** loops, and are referred together as a *time-stamp* of the segment.

Similarly, we denote a communication segment generated by the compiler for a reference pattern $P : \ulcorner a(\delta_{k+1}, \ldots, \delta_n) \Rightarrow (i_{k+1}, \ldots, i_n) : D \mid \gamma \urcorner$ by $\mathsf{C}(a, P, D, \delta_1, \ldots, \delta_k)$, where $(\delta_1, \ldots, \delta_k)$ is its time-stamp.

Any point between two computation segments is a potential location for a communication segment. However, not every such point is legal. A communication should happen no earlier than the time when the referenced data is ready and no later than the time when the data is to be used. We define the notion of a *communication window* to specify the range within which a communication segment can be inserted between any two computation segments.

Given a communication segment, $\mathsf{C}(a, P, D, \delta_1, \ldots, \delta_k)$, the *top* of its communication window is the point immediately after the last of the set of computation segments including $\mathsf{S}(a, D, \delta_1, \ldots, \delta_k)$ and those which compute the indirect array references occurring in $\ulcorner \delta_1, \ldots, \delta_n \urcorner$. The *bottom* of the window is the point immediately before the earliest of the set of computation segments in which $a(i_1, \ldots, i_n)$ is used.

Example 6.4 Consider a reference pattern $P : \ulcorner a(i, j, t) \leftarrow b(i, c(i, j, t), t - 1) \urcorner$. The top of the communication window for communication segment $\mathsf{C}(b, P, D, t - 1)$ is the point immediately after computation segments $\mathsf{S}(b, G, t - 1)$ and $\mathsf{S}(c, G, t)$. Since the time-stamp of $\mathsf{S}(c, G, t)$ is newer, the top is the point right after it.

Example 6.5 Consider the shared-memory program in Figure 6.3. The following spatial reference patterns can be derived from it:

$$P_1' : \ulcorner a@(i, x) \Rightarrow (i, j) : D \mid j = t \text{ and } 1 \leq x \leq n \urcorner,$$

```
dom T = [0..n];
dom D = [1..n] × [1..n];
for (t : T) {
  forall ((i, j) : D)
    b(i, j, t) = if (j = t) then \ + {a(i, x, t − 1) | 1 ≤ x ≤ n}
                 || else 1
                 fi;
  forall ((i, j) : D)
    a(i, j, t) = if (t = 0) then 0
                 || (i = b(1, t, t)) then a(i, j, t − 1)
                 || else b(i, t, t)
                 fi;
}
```

Figure 6.3: A shared-memory program.

$$P_2' : \ulcorner b@(1,t) \Rightarrow (i,j) : D \mid t \neq 0 \urcorner,$$

$$P_3' : \ulcorner a@(i,j) \Rightarrow (i,j) : D \mid t \neq 0 \text{ and } i = b(1,t,t) \urcorner,$$

$$P_4' : \ulcorner b@(i,t) \Rightarrow (i,j) : D \mid t \neq 0 \text{ and } i \neq b(1,t,t) \urcorner.$$

Corresponding to these reference patterns, three communication segments can be defined. P_3' corresponds to a local memory access, hence can be ignored. With respect to these segments, the following communication windows can be derived:

Comm. Segment	*Communication Window*	
$\mathsf{C}(a, P_1', D, t-1)$	$\mathsf{S}(a, G, t-1)$	$\mathsf{S}(b, G, t)$
$\mathsf{C}(b, P_2', D, t)$	$\mathsf{S}(b, G, t)$	$\mathsf{S}(a, G, t)$
$\mathsf{C}(b, P_4', D, t)$	$\mathsf{S}(b, G, t)$	$\mathsf{S}(a, G, t)$

Note that a communication window may cross loop iterations, which is the consequence of cross-iteration dependences.

Remarks

Consider the case where a processor computes some values and then sends them to other processors. The processor can either compute and send one value at a time or it can compute many values first and then send them all at once. The difference is the memory usage (data needs to be stored if it doesn't get sent) and the communication overhead (more frequently, small messages incur more overhead such as message startup time and time for the calls to the operating system kernel). With respect to the above example, the inner multiple loops can be broken down to smaller segments.

Example 6.6 The following three multiple loops are possible decompositions of computation segment $\mathsf{S}(a, G, t)$ (assume $G = G_1 \times G_2$):

forall $(i : G_1)$	forall $(j : G_2)$	forall $((i,j) : G)$
$\mathsf{S}(a, G_2, t, j)$;	$\mathsf{S}(a, G_1, t, i)$;	$\mathsf{S}(a, \mathsf{nil}, t, i, j)$;

Segments $\mathsf{S}(a, G_2, t, j)$, $\mathsf{S}(a, G_1, t, i)$, and $\mathsf{S}(a, \mathsf{nil}, t, i, j)$ are all of smaller granularity than $\mathsf{S}(a, G, t)$.

Selecting the appropriate granularity of computation segments and consequently the size and frequency of message-passing requires cost-driven optimization based on both the target machine parameters and the cost estimation of the program. The formulation here provides a framework for doing so. Our method uses both computation segments and communication segments at their maximum granularity.

6.4.2 Two Simple Scheduling Strategies

Given a communication window, what is the best placement of a communication segment? This problem involves trade-offs in communication cost, memory usage, balanced network flow, and so on, and needs to be answered by cost-driven optimizations based on a model of target machine characteristics. We describe a scenario

```
for (t : T) {
  S(b, G, t);
  C(b, P'_2, D, t);
  C(b, P'_4, D, t);
  S(a, G, t);
  C(a, P'_1, D, t);
}
```

(a) Simple strategy 1

```
for (t : T) {
  C(a, P'_1, D, t − 1);
  S(b, G, t);
  C(b, P'_2, D, t);
  C(b, P'_4, D, t);
  S(a, G, t);
}
```

(b) Simple strategy 2

Figure 6.4: Two simple communication scheduling strategies.

here to illustrate the trade-off between processor idling time versus network message traffic.

A processor waiting for a message cannot proceed with its own program until the message is received. So the earlier the message is sent out by the processor which produces the required data, the better. On the other hand, if the production and consumption of the messages are too much off-balance, messages may start to saturate the network. Profiling and estimating computation time, message size, etc., should provide clues for where the communication segment should be placed to maintain a smooth flow of message traffic.

Consequently, we want to put communication segments in places where message aggregation can be performed. However, larger messages also mean larger buffers. For large applications, this could cause a shortage of memory. Again, to balance the issue, cost estimation and profiling are needed.

Here we propose two very simple strategies which do not take cost into consideration:

1. Place a communication segment at the top of its communication window.

2. Place a communication segment at the top of its communication window if the

top is a computation segment with the same time-stamp; otherwise place it immediately before the first computation segment in the window that has the same time-stamp.

Using the first strategy on the program shown in Figure 6.3, we obtain a schedule shown in Figure 6.4(a); using the second strategy, the result is shown in Figure 6.4(b).

The first strategy has an advantage in controlling granularity, since *strip-mining* (cf. [Wol89]) can be applied to a computation segment and the adjacent communication segments. The second strategy is slightly easier for generating code, since there are no cross-iteration dependences between computation segments and communication segments.

6.5 Synchronizing Communication Routines

For the purpose of discussing message synchronization, communication routines are categorized into two groups: *uniform routines* and *aggregate routines.* Uniform routines include Shift, Uniform-Shift, Copy, and Send-Receive (the last is a special case). We illustrate each case with examples.

6.5.1 Synchronizing Uniform Routines

As mentioned in Chapter 2, there are two special forms of a communication pattern, the *sender's form* and the *receiver's form.* For a uniform routine, the critical issue in synchronization is to derive both the sender's form and the receiver's form of a communication pattern.

Inverting a Communication Pattern The sender's form and the receiver's form of a communication pattern

$$\ulcorner a@(\sigma_1, \ldots, \sigma_n) \Rightarrow (\delta_1, \ldots, \delta_n) : D \mid \gamma \urcorner,$$

can be expressed as

$$\text{Sender's form: } \ulcorner a@(i_1,\ldots,i_n) \Rightarrow (\delta'_1,\ldots,\delta'_n) : D \mid \gamma \urcorner,$$
$$\text{Receiver's form: } \ulcorner a@(\sigma'_1,\ldots,\sigma'_n) \Rightarrow (i_1,\ldots,i_n) : D \mid \gamma' \urcorner.$$

Tuples $(\sigma'_1,\ldots,\sigma'_n)$ and $(\delta'_1,\ldots,\delta'_n)$ are related in the following way. Suppose we can write the source and destination expressions as

$$(\delta'_1,\ldots,\delta'_n) = T_1(i_1,\ldots,i_n),$$
$$(\sigma'_1,\ldots,\sigma'_n) = T_2(i_1,\ldots,i_n)$$

where T_1 and T_2 are well-defined functions. Then T_1 and T_2 must be inverses of each other.

When $(\sigma'_1,\ldots,\sigma'_n)$ and $(\delta'_1,\ldots,\delta'_n)$ are linear expressions of the indices, it is possible to determine symbolically the sender's and receiver's forms. But in general, a compiler would not be able to do so. Our restriction on the array index expressions on the left-hand side of an assignment statement (the second assumption on the shared-memory program form) is to assure that at least the receiver's form is readily available to the compiler.

Every uniform routine in Table 6.1 and 6.2 corresponds to a communication pattern that can be transformed symbolically into both sender's and receiver's forms by a compiler. We show the synchronization process through an example.

Example 6.7 Consider the multiloop nest in Figure 6.5 (a). First, a spatial reference pattern, P : $\ulcorner a@(i+1, j-2) \Rightarrow (i,j) : D \mid t > 1 \urcorner$, is derived. Then the pattern is matched with a uniform routine Uniform-Shift$(D,(-1,2),a,a_1,B)$. The target code is shown in Figure 6.5 (b), where C(a,P,D,t) includes a call to the routine.

The uniform communication routine Uniform-Shift$(D,(-1,2),b)$ is actually implemented by a pair of send and receive commands, which are executed on every processor participating in the communication:

$$\text{send}(D,(-1,2),a,B);$$
$$\text{receive}(D,(1,-2),a_1,B).$$

```
for (t : T)
  forall ((i, j) : D)
    b(i, j, t) = if (t > 1) then
                   a(i + 1, j − 2, t)
                 || else b(i, j, t − 1)
                 fi;
```

(a)

```
forall ((p, q) : E)
  for (t : T) {
    if (t > 1) then C(a, P, D, t);
    S(b, G, t);
  }
```

(b)

Figure 6.5: Synchronizing uniform communication routines.

The send statement is derived from the sender's form of the reference pattern while the receive statement is derived from the receiver's form. Since the source program already contains the receiver's form, the compiler only needs to derive the sender's form

$$\ulcorner a@(i,j) \Rightarrow (i-1, j+2) : D \mid t > 1 \urcorner.$$

The derivation involves a matrix inversion. Under the condition that the matrix is full rank (which is met by all **Uniform-Shift** cases), the inversion can be computed symbolically.

A pair of **send** and **receive** commands is always arranged as a nonblocking **send** followed by a blocking **receive** in the program. The predicates for the **send** and receive commands are arranged in such a way that for every message sent out to the network, there is a receiving statement matching it.

6.5.2 Synchronizing Aggregate Routines

The synchronization issue for an aggregate communication routine is to make sure that the routine is invoked under the same condition on all the participating processors, so that they will all reach whatever primitive communication commands that implement the routine.

As we mentioned in Section 6.3, predicates in a reference pattern are ignored in the process of matching communication patterns. However, for synchronizing an aggregate communication routine, predicates must be used.

Lifting Space-Invariant Predicates A Boolean predicate P of a reference pattern is said to be *space-invariant* with respect to a data-layout strategy if the value of P is invariant with respect to the values of the spatial indices, otherwise, it is said to be *space-variant.*

For example, suppose indices (i, j, t) are defined over domain $D \times T$ where D is partitioned. Then predicate $(i > j)$ is space-variant since for different values of i and j, $(i > j)$ can have different values. On the other hand, predicate $(t > 1)$ is a space-invariant predicate.

When a space-invariant predicate is in conjunction with a space-variant predicate, as in $(t > 1)$ and $(i > j)$, it is lifted outside of the call to the communication routine while the space-variant predicate is ignored. Since the space-invariant predicate will evaluate to the same value for all participating processors, all, or none, of the processors will participate in the communication, as shown in the following example:

Example 6.8 Consider the multiloop nest in Figure 6.6 (a). A spatial reference pattern is derived from it:

$$P : \ulcorner a@(3, j) \Rightarrow (i, j) : D \mid t > 1 \text{ and } i > j \urcorner.$$

It is then matched with a $\mathsf{Spread}(D, 1, 3, a, a_1, B)$, by ignoring the predicate $i > j$. The message-passing code is shown in Figure 6.6 (b), where $\mathsf{C}(a, P, D, t)$ includes a call to $\mathsf{Spread}(D, 1, 3, a, a_1, B)$.

6.5.3 Proof of Deadlock-Free

We prove that the compiler does not introduce any deadlock during the communication generation process.

```
for (t : T)
  forall ((i, j) : D)
    b(i, j, t) = if (t > 1) and (i > j)
                 then a(3, j, t)
                 || else b(i, j, t − 1)
                 fi;
```

(a)

```
forall ((p, q) : E)
  for (t : T) {
    if (t > 1) then C(a, P, D, t);
    S(b, G, t);
  }
```

(b)

Figure 6.6: Synchronizing aggregate communication routines.

Recall that the target code generated by the compiler is in the SPMD style. It consists of a sequence of multiple loops, each with a sequence of computation and communication segments as its loop body.

Assume that each computation segment is a single-entry single-exit segment (i.e. there are no **goto** or **break** statements), and is generated by the compiler based on semantics-preserving transformations which do not introduce deadlock. Provided that the source program is correct and all the data that a computation segment requires are available, then its execution always terminates. Therefore, we only need to check the behavior of communication segments which ensure the availability of the data required by the computation segments. We prove, by induction on the sequence of communication segments, that the communication segments of a loop do not introduce any deadlock.

Induction Hypothesis: The program is deadlock-free up to the $(N-1)$th communication segment.

Proof:

By the assumption on the computation segment and the induction hypothesis, each processor will reach the beginning of the Nth communication segment. Now we discuss two cases:

Case 1: The communication segment consists of a uniform communication routine. Since such a communication routine is guarded only by space-invariant predicates, all processors will execute the routine. Since the communication routine is assumed to terminate, the entire segment terminates.

Case 2: The communication segment consists of an aggregate communication routine. We assume that the message buffer is large enough to hold the entire data transmitted in a message.[3] According to the arrangement of **send** and **receive** statements, every processor executes a **send** statement before a **receive** statement. Since no deadlock would occur due to buffer overflow, every processor entering the communication segment will eventually finish executing the **send** statement, and move on to the **receive** statement. Finally, because the predicates for the **send** and **receive** statements match with each other, every **receive** statement will terminate with received data. Therefore, the Nth communication segment eventually terminates, and so the program also terminates. □

6.6 Discussions

Synchronizing a uniform routine depends on computing the inverse of a reference pattern. In the event that the inverse is not computable at compile-time, our current solution is to use an aggregate routine instead. Such a routine requires every member in a well-defined subset of the network of processors (such as a column) to participate, including those that do not really need the data. This simple solution may incur a high cost in some cases. For example, suppose the reference pattern

$$\ulcorner a@(2,j) \Rightarrow (c(i,j),j) : D \urcorner$$

contains an indirect reference $c(i,j)$ whose value cannot be determined at compile-time. Our pattern-matching algorithm would match it with a **Spread**, but a **Copy** would suffice if $c(i,j)$ was known to be constant at compile-time.

[3]This assumption can be relaxed if we take buffer size into consideration when generating communication.

Asynchronous Communication An alternative approach is to generate a request and receive pair which interrupts the processor holding the requested value. The target program looks like

```
forall (p : E) :
  if (i = 2) then {
    request(idx_to_pid(c(i,j),j),a);
    receive(idx_to_pid(c(i,j),j),a);
}
```

The request and receive pair works as follows: Whenever there is a request coming to a processor, an interrupt handler will send out the requested data if it is ready, otherwise it will queue the request and send out the value when it becomes available. However, this could be very slow due to frequent context switching.

User Directives Another alternative is to allow the user to provide enough information to generate efficient communication. It turns out that all that is needed are two functions that are the inverses of each other for specifying the sender's form and the receiver's form of a given reference pattern. Using the same example shown above, the user can say the following:

Communication Forms:

$T(i,j) = (c(i,j),j) = \{(i \text{ div } j, j)\}$

$T\_inv(i,j) = \text{if } (i <= (n \text{ div } j)) \text{ then}$

$\{(k,j) \mid i * j <= k < \min(n+1, (i+1) * j)\};$

The inverse T_inv can then be used to generate a send and receive pair for efficient communication. The corresponding target code will look like

```
forall (p : E)
  if (i = 2) then
```

send(idx_to_pid($T(i,j)$));

if ($p \in$ {idx_to_pid($T\_inv(i,j)$)}) then

receive(idx_to_pid($2,j$));

Chapter 7

Experiments

Several programs have been successfully compiled using the Crystal compiler. The target machines include an Intel iPSC/1 (with 32 nodes), an Intel iPSC/2 (with 64 nodes), an NCUBE/7 (with 128 nodes) and an NCUBE/10 (with 512 nodes). Results from these machines indicate that the performance of the compiler-generated code is comparable to those of manually written ones. In this chapter, we present some preliminary performance results and discuss some performance-related issues. In Section 7.1 a brief description of the experimental compiler is given. The performance study method is presented in Section 7.2. Finally, concrete benchmark results for a group of applications are shown in Section 7.3.

7.1 The Experimental Compiler

The experimental Crystal compiler is written in T [RA82,RAM88], a dialect of scheme [SS78]. Currently, the compiler can generate code for the iPSC/2 and nCUBE I hypercubes. In both cases, the compiler generates a C program to be executed on each node processor. The program is extended with calls to communication routines that are preimplemented on the specific target machine.

To run the compiler, the user first invokes a T process, then loads the Crystal

compiler into T. The compilation process is broken into eight steps: (1) parsing; (2) preprocessing; (3) dependence analysis; (4) index domain alignment; (5) control structure synthesis; (6) communication analysis; (7) intermediate code generation; and (8) target code generation. The user can choose to compile a program step by step, and examine the output of each step. All the intermediate results can be directed to a log file.

The experimental Crystal compiler has many limitations. First of all, with respect to the topics covered by this dissertation, the following features are not implemented: while-active loop, code refinement, block-linking by independent distribution and by replication. Secondly, the compiler requests extra information to be input by the user. This compiler is written for an old version of Crystal, in which index domains are not explicitly declared. Thus the user has to provide the index domain declaration and data type information separately. Thirdly, the programming environment is very primitive. There is no debugger or on-line help manual. Finally, the compiler is still quite buggy.

7.2 Performance Study Method

7.2.1 Performance Measurement

The performance of each compiler-generated program is measured by two related parameters: *elapsed time* and *speedup*.

Elapsed Time (T) To measure the total elapsed time of a node program, timing statements are inserted into the program generated by the compiler. The starting point of the timing is the first executable statement in the node program, and the ending point of the timing is after the last executable statement.

The elapsed-time of an application can be divided into three parts:

- T_i — Time for loading data into the nodes.

- T_p — Time for the essential computation using only the local copy of the data on each node. This is the time that would be needed on a uniprocessor.

- T_c — Time for interprocessor communication.

To measure these portions, we put timing statements around computation segments and communication segments to get T_p and T_c, and around initialization statements to get T_i.

Speedup (S) Speedup is defined as the ratio of the parallel execution time to the sequential execution time. It shows how an application program scales with the size of a parallel machine.

Let T_1 denote the elapsed time of a program running on a single processor, and T_k be that on k processors. The speedup is computed as

$$S = \frac{T_1}{T_k}. \tag{7.1}$$

7.2.2 Analysis and Comparisons

The goal of the performance study is to see how realistic the compiler techniques and the compiler organizations are. We have conducted the study in several directions.

Comparison with Hand-Written Programs To see how the compiler-generated code performs comparing to hand-crafted code, for several applications, we have obtained independently written, hand-crafted programs. The performances of two programs for the same application are measured under the same set of parameters. The results show that the performance of the compiler-generated code is within a factor of 2 or 3 of that of the hand-crafted code.

Effects of Code Refinement To see where the slowdown factors are in the compiler-generated code, with two applications we have conducted studies on the effects of code

refinement. Several refinement techniques are manually introduced to the target program and turned on one by one, and program performance is measured for each case. The results show that introducing multiple-assignment and lifting predicates out of loops are the two most effective refinements, and together they can reduce the elapsed time of a program by more than 50%.

Comparison with Programs Using Asynchronous Communication The communication routines in the compiler-generated programs are so-called *loosely synchronous routines*, in which the sender and the receiver of a message must synchronize with each other. Using loosely synchronous routines may sometimes cause delays or extra messages. In Chapter 6 we discussed some alternatives to this approach. Among them is the *asynchronous communication approach*, in which each message is sent out at the time it is needed, and it will interrupt the receiver when it arrives. With one application, we have conducted a comparative study of synchronous and asynchronous communication. The results show that the overhead of asynchronous communication is very high; hence, overall using loosely synchronous routines is still better.

7.3 Application Benchmarks

7.3.1 Matrix Multiplication

The Crystal program MM (Figure 2.1) computes the product of two $n \times n$ matrices, a and b, using the standard matrix multiplication algorithm.

The target code for node processors generated by the compiler consists of three parts: initialization, broadcast, and computation. Two partition parameters, p_1 and p_2, are input at runtime, representing a (p_1, p_2)-block layout strategy for both matrices a and b. Initially, each processor holds two submatrices a_{sub} and b_{sub} of size $n/p_1 \times n/p_2$.

The program runs in two steps. In the first step, each processor spreads its a_{sub} to

Machine	*Matrix Size*	1	2	4	8	16	32	64
iPSC/1	50 × 50	9.4	4.6	2.5	1.4	1.0	0.5	–
	100 × 100	69.9	36.4	18.0	9.0	4.8	2.5	–
NCUBE/7	50 × 50	6.3	3.1	1.5	0.8	0.5	0.3	0.2
	100 × 100	–	–	12.1	5.9	3.2	1.6	1.0

(a) Elapsed time on iPSC/1 and NCUBE/7 (in seconds).

Matrix Size	1	2	4	8	16	32	64
100 × 100	10.8	5.5	2.7	1.4	0.7	0.4	0.2
128 × 128	22.7	11.3	5.7	2.9	1.5	0.8	0.5
200 × 200	89.3	43.7	21.7	10.9	5.5	2.8	1.4
256 × 256	184.2	92.9	46.1	22.9	11.4	5.7	3.0
300 × 300	299.1	148.4	73.9	37.3	18.4	9.4	4.8

(b) Elapsed time on iPSC/2 (in seconds).

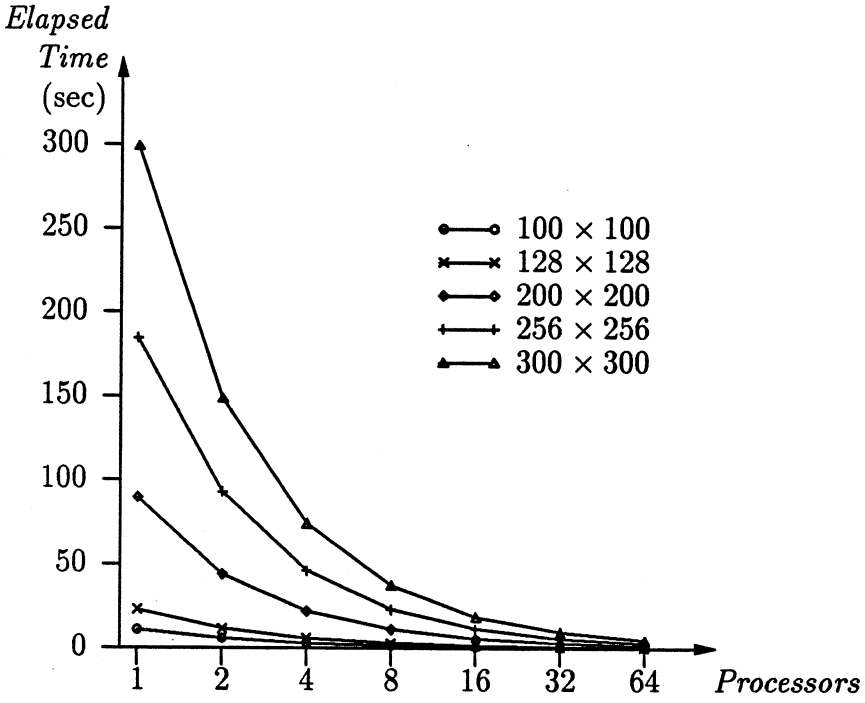
P	1	2	4	8	16	32	64
Matrix Size	100	141	200	283	400	566	800
T	10.8	15.3	21.7	31.3	43.8	62.7	87.7

(c) Elapsed time with fixed problem size per node on iPSC/2 (in seconds).

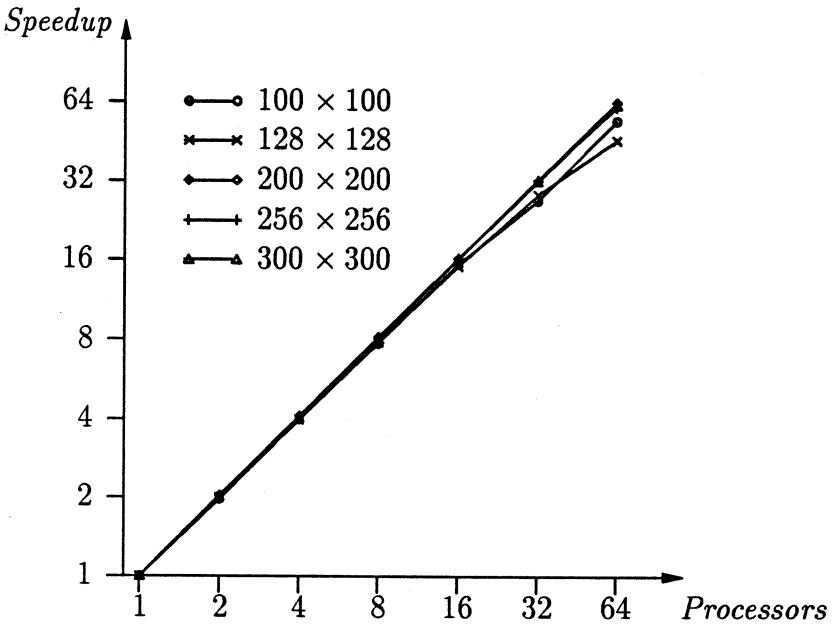
Table 7.1: Performance of the MM program.

processors in the same row of the processor network, and spreads its b_{sub} to processors in the same column. These data spreadings are done by the communication primitive Spread. Each processor ends up having a_{row} and b_{col}, which contain n/p_1 rows of A and n/p_2 columns of B, respectively. In the second step, each processor computes the product of a_{row} and b_{col}.

Tables 7.1(a) and (c) and Figures 7.3.1(a) and (b) show the performance of the compiler-generated program. The experimental data is obtained as follows: the problem size (i.e. matrix size) is first fixed. The same problem is then run on different numbers of processors, from one to 64.



(a) Elapsed time on iPSC/2.



(b) Speedup.

Figure 7.1: Performance of the MM program.

7.3.2 Gaussian Elimination with Partial Pivoting

The Crystal program Gauss (Figure 3.4 in Section 3.1) implements an algorithm for Gaussian elimination with partial pivoting. The program iterates over the columns of the input matrix. In iteration k a pivot element is chosen from the elements in column k at or below the diagonal (say element (j, k)) and rows k and j are exchanged. Then, the elements in the column below the diagonal are eliminated using the pivot element.

Tables 7.2(a)–(d) and Figures 7.2(a)–(b) show the performance of the compiler-generated program. The experimental data is obtained as follows: the problem size (i.e. matrix size) is first fixed. The same problem is then run on different numbers of processors, from one to 64. When running on more than one processors, column partition of the matrix is used.

The speedup of program Gauss (Figure 7.2(a)) is not terrific. As the number of processors increases, the amount of computation in each processor decreases linearly. The communication time, in spite of the decrease in the size of the messages, stays roughly the same since the cost of broadcasting a message of some unit size increases with the increasing number of participating processors. Consequently, the overhead of communication (with respect to some unit computation) increases, and results in a speedup that is far from being linear.

Tables 7.2(c) and 7.2(d) show the effects of various code refinements that are discussed in Chapter 3. The combined effect of these refinements reduces the total elapsed time by more than half. To see how each refinement contributes to the reduction, a control experiment is conducted. In the experiment, refinements are turned on one by one, and timing is measured after each new refinement is added in. The results (Table 7.2(d)) show that introducing multiple assignments and lifting predicates out of loops are the two most important refinements.

Table 7.2(b) and Figure 7.2(b) contain the comparison of the performance of the compiler-generated program Gauss and a hand-crafted program for the same application on an iPSC/2 hypercube. Gauss is obtained with all the optimization

N	1	2	4	8	16	32	64
100	5.4	3.2	2.2	1.8	1.7	1.7	1.8
128	10.6	6.0	3.9	2.9	2.5	2.5	2.5
200	38.3	20.8	12.7	8.1	6.3	5.6	5.4
256	77.8	41.4	23.4	14.8	10.7	8.9	8.3

(a) Elapsed time on iPSC/2 (in seconds).

N	*Programs*	1	2	4	8	16	32	64
64	Hand-Written	0.51	0.41	0.32	0.34	0.39	–	–
	Compiler	1.47	0.96	0.72	0.63	0.62	–	–
	Ratio	2.9	2.3	2.2	1.9	1.6	–	–
128	Hand-Written	3.82	2.72	1.63	1.14	1.07	1.11	1.21
	Compiler	10.08	5.75	3.64	2.65	2.21	2.05	2.11
	Ratio	2.6	2.1	2.2	2.3	2.1	1.9	1.7

(b) Comparisons of elapsed time on iPSC/2 (in seconds).

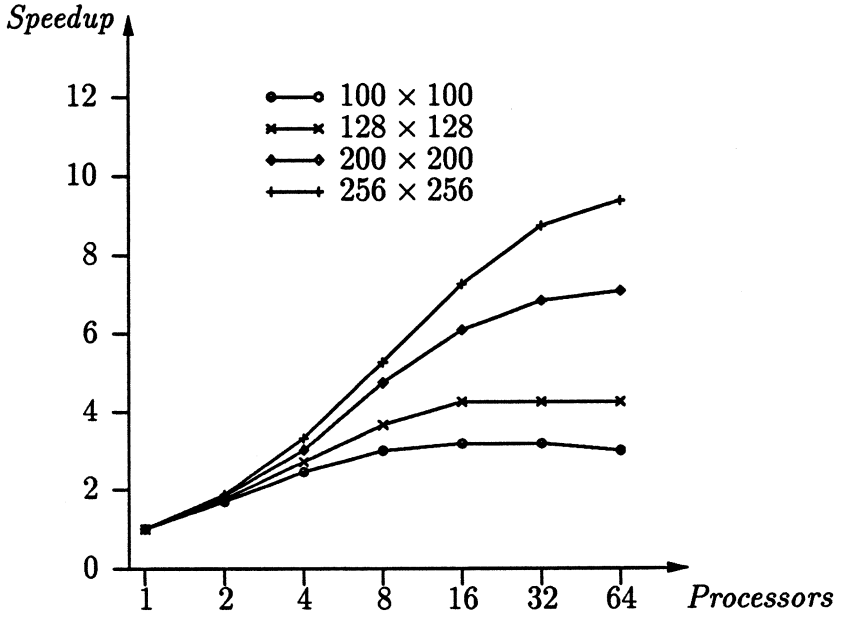
Programs	*Time*	*Ratio*
Compiler code, no refinement	17.0	4.9
Compiler code, with refinement	8.2	2.3
Hand-written code	3.5	1

(c) Effects of code refinement on iPSC/2. $M = 256 \times 256$, $P = 32(1 \times 32)$.

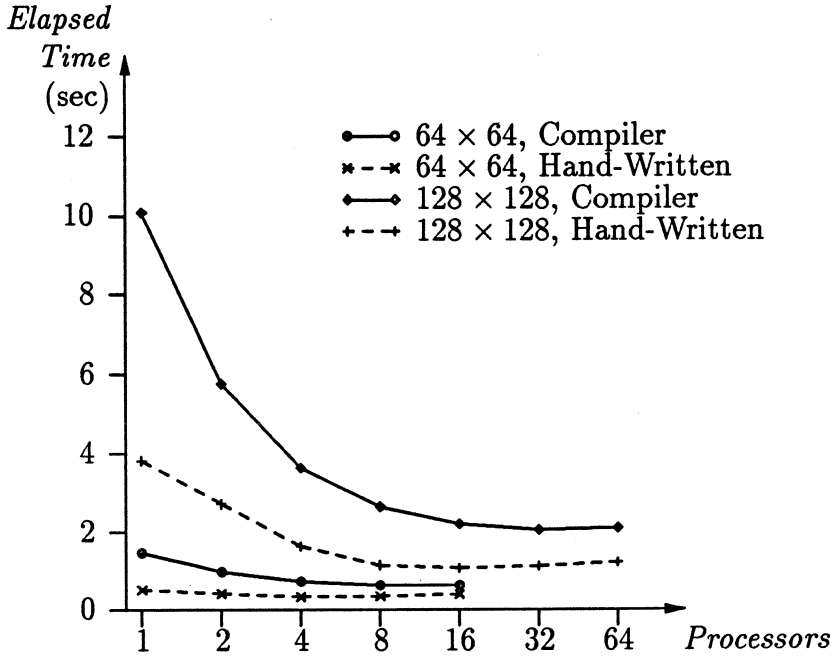
Code Refinement	*Time*	*Normalized*
None	17.0	100 %
Introduce Multi-Assigns	12.5	73.5 %
Lift Preds out of Loops	8.6	50.6 %
Eliminate Sub-Exprs	8.5	50.0 %
Reduce Buffer Copying	8.2	48.2 %

(d) Effect of each refinement step.

Table 7.2: Performance of the Gauss program.



(a) Speedup.



(b) Comparisons of elapsed time on iPSC/2.

Figure 7.2: Performance of the Gauss program.

switches turned on. The hand-crafted program is written directly by an experienced programmer familiar with the iPSC/2 system. Figures given in the table indicate the total elapsed time in *seconds* of a program running on different numbers of processors, with a fixed problem size.

A factor of 2 to 3 inefficiency in the compiler generated program can be attributed to two reasons. One is that the compiler-generated program is more flexible and works for different data-layout strategies (e.g. row-, column-, or block-partition). The hand-crafted version, however, is optimized for a particular partitioning strategy. The compiler can be improved to produce more efficient code in this respect by performing partial evaluating and optimizing the target code once a specific partitioning strategy is chosen. The other factor is in the optimization of the sequential code. The hand-crafted program uses pointers extensively while the compiler-generated code tends to do more data copying. This indicates that a parallelizing compiler must also be good at optimizing sequential code.

7.3.3 Connected Components of a Graph

The Crystal program Ccomp (Figure 5.7 in Chapter 5) implements the *mesh-of-trees connected components algorithm* [Ull84](pages 160-164). The input to the algorithm is the adjacency matrix of a directed graph, and the output is an assignment of group numbers to the nodes in the graph. The overall outline of the algorithm is as follows. Initially each node is in a group by itself. Repeatedly, each group g finds the lowest-number group h to which it is adjacent, in the sense that there is an edge from some node in g to some node of h. If h is lower than g, g is merged into h, by giving the nodes of g the same number as h. After $\log(n)$ iterations, every group is a connected component by itself.

The Crystal program consists of four data fields: $comp(v, i)$ represents the component of node v at stage i, which we take to be the smallest index of any node in that component; $min\_nbr(v, i)$ represents the minimum value of $comp(u, i)$ over all nodes

u adjacent to v; $min\_comp(v,i)$ represents the minimum value of $comp(u,i)$ over all nodes u adjacent to component v at stage i, including vertex v itself; and finally, $next(v,i,\log n)$ represents the transitive closure of $min\_comp(v,i)$, i.e. the minimum value of $comp(u,i)$ over all components reachable from v at stage i; we know this is computable in at most $\log(n)$ steps, and hence value of the third index j.

Tables 7.3(a)–(c) and Figures 7.3.3(a)–(b) show the performance of the compiler-generated code. Table 7.3(a) lists the elapsed time of the program with three sets of data. In each case, a random graph with the specified features is generated. Since the program is very communication-bounded, it does not have a very good speedup. In fact, in the first two cases, we observe some "speed-down."

Table 7.3(b) and Figure 7.3.3(a) show the detailed timing results. In all three cases, the computation time T_p decreases linearly with the increase in the number of processors. However, the communication time T_c, which represents the time the program spends on several broadcasting routines, increases with the number of processors.

The communication required by this algorithm does not have regular patterns. Using loosely synchronous routines to implement them, as the compiler does, causes extra data to be passed around in the network. To confirm our belief that loosely synchronous routines may behave just as well as other alternatives, we compared the performance of the compiler generated code with that of an independently written program using asynchronous communication (i.e. interruptions and handlers). The results are shown in Table 7.3(c) and Figure 7.3.3(b). The overhead associated with the asynchronous communication is very high, making the overall performance of the asynchronous program unsatisfying. When running with large numbers of processors, the asynchronous program encounters some unpredictable communication errors, which cause the program to hang. A detailed study on this subject can be found in [Hu91].

Parameters	1	2	4	8	16	32	64
$V = 64, E = 100$	0.26	0.19	0.16	0.15	0.15	0.16	0.18
$V = 128, E = 200$	1.03	0.64	0.46	0.36	0.33	0.33	0.34
$V = 256, E = 200$	4.34	2.39	1.44	1.00	0.79	0.69	0.65

(a) Elapsed time (in seconds).

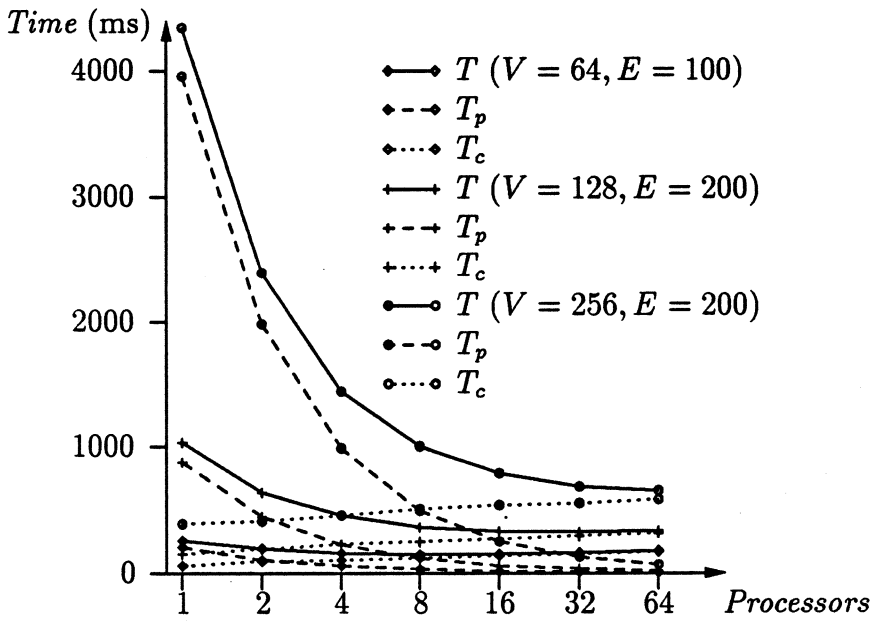
Parameters	*Timing*	1	2	4	8	16	32	64
$V = 64$,	T_p	203	102	56	30	15	11	6
$E = 100$	T_c	55	91	104	117	137	152	172
$V = 128$,	T_p	879	447	225	115	59	33	18
$E = 200$	T_c	152	188	232	249	273	295	320
$V = 256$,	T_p	3954	1980	985	493	249	128	67
$E = 200$	T_c	388	408	455	510	541	557	586

(b) Classified timings (in milliseconds).

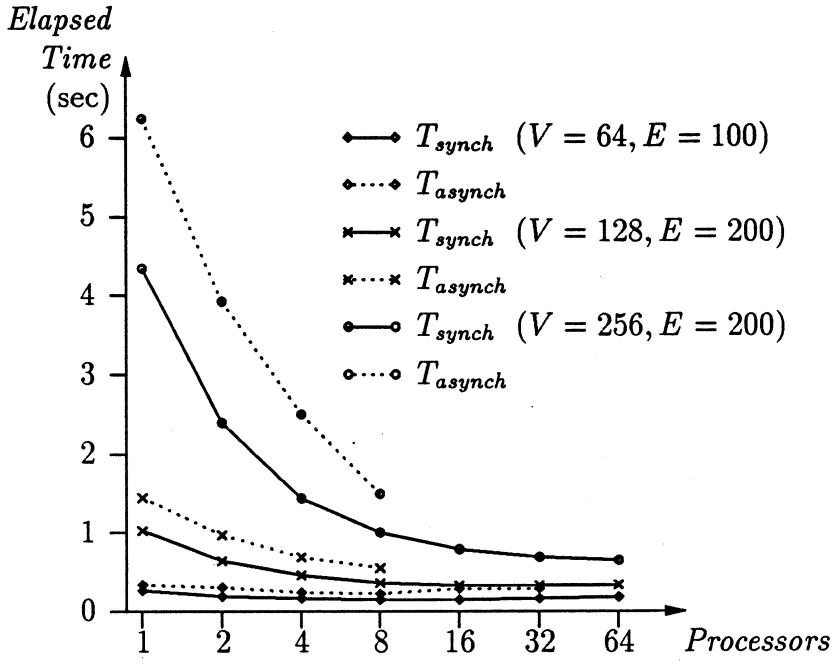
Parameters	*Program*	1	2	4	8	16	32	64
$V = 64$,	Synch.	0.26	0.19	0.16	0.15	0.15	0.16	0.18
$E = 100$	Asynch.	0.34	0.30	0.24	0.22	0.29	0.29	—
$V = 128$,	Synch.	1.03	0.64	0.46	0.36	0.33	0.33	0.34
$E = 200$	Asynch.	1.45	0.97	0.69	0.55	—	—	—
$V = 256$,	Synch.	4.34	2.39	1.44	1.00	0.79	0.69	0.65
$E = 200$	Asynch.	6.24	3.92	2.50	1.50	—	—	—

(c) Comparisons of synch.- and asynch.-comm. (in seconds).

Table 7.3: Performance of the Ccomp program on iPSC/2.



(a) Classified timings.



(b) Comparisons of synchronous and asynchronous code.

Figure 7.3: Performance of the Ccomp program.

7.3.4 A Financial Application

The Crystal program Finance implements a financial application for evaluating mortgage-backed securities. The algorithm is designed to determine the "fair" value of a pool of mortgages for investors who collectively share the uncertainty in interest rates and the risk of prepayments. The method of analyzing a pool of mortgages involves the simulation of the behavior of uncertain monthly interest rates for thirty years and the corresponding cash flows generated by a pool of mortgages for each interest rate scenario.

This program is highly computation-intensive, but also highly parallelizable. The C code generated by the Crystal compiler contains several reduction operations for computing averages and covariances. Each reduction operation collects and combines results from a set of nodes to a single node. For this application, due to its large size (several Mbytes) the historical interest rates are stored on the disk and, at execution time, read into main memory on the host and distributed to the nodes so that each node gets a portion of the data. The host program is currently written manually.

The speedup results of the compiled parallel C code are given in the tables. Table 7.4(a) summarizes the performance figures for the 512 node, 512 Kbyte memory per node NCUBE/10 at Caltech, and Table 7.4(b) for the 64 node, 4Mbyte memory per node Intel iPSC/2 with floating point unit at Yale. All times are in seconds.

The tables show two different definitions for speedup calculation. Both speedup figures are calculated according to Equation 7.1. However, in calculating S_1, the parallel execution time T_{par} includes both the computation time and the data loading time. Everything except those portions of the program that perform essential computation is considered as overhead, and we are ignoring the reduction time required in a uniprocessor implementation. This definition of speedup was chosen because the memory limitation on our processors prevents us from running the entire program on one processor, which is required for the conventional definition of speedup. In shared-memory multiprocessor machines, using only one processor gives that processor the

P	T	$(T_i,$	$T_p,$	$T_c)$	S_1	(S_2)
512	198.8	(37.0,	134.3,	27.5)	345.9	(419.2)
256	191.1	(31.5,	139.8,	19.8)	187.3	(224.3)
128	186.6	(28.8,	147.5,	10.3)	101.2	(119.6)

(a) Scaled-speedups for fixed problem size per node on NCUBE/10.

P	T	$(T_i,$	$T_p,$	$T_c)$	S_1	(S_2)
64	128.8	(14.3,	108.9,	5.5)	54.1	(60.8)
32	126.3	(12.5,	109.5,	4.2)	27.8	(30.8)
16	122.7	(9.1,	110.2,	3.3)	14.4	(15.5)

(b) Scaled-speedups for fixed problem size per node on iPSC/2.

Table 7.4: Performance of the Finance program.

whole memory, but for distributed-memory machines, turning the other processors off results in their memory being inaccessible.

The second speedup figure (S_2) gives a more insightful and more optimistic view of the performance and is calculated by excluding data loading time from T_{par}. It is our observation that for small problem sizes (e.g. two-year simulations), the data loading time dominates for runs with large numbers of processing elements. However, for large problem sizes (30-year simulations for the cases in the tables), the data loading time does not play a big role in total execution time.

Chapter 8

Conclusions

8.1 Summary of Contributions

This dissertation has made the following major contributions to the field of compiling high-level languages to distributed-memory machines:

1. It provides an elegant formulation of and a practical solution to the index domain alignment problem, and it recognizes that domain alignment is a necessary step preceding data layout in a compiler for distributed-memory machines.

2. It provides a novel compilation technique for generating communication statements. Using this technique, the compiler can generate not only the simple send-and-receive commands, but also more sophisticated routines like shifting, broadcasting and reduction, thus making communication more efficient.

3. It provides a novel algorithm for transforming a functional program into an explicit parallel program.

4. It presents a complete compiler design for a high-level language for distributed-memory machines. The compilation model is based on source-level program transformations and optimizations, and can be adapted to different source languages and different target machines.

5. It demonstrates, through the implementation of an experimental compiler, that both the individual compilation techniques and the overall compilation design are practically sound.

Array allocation often has a critical impact on the overall performance of a program on a distributed-memory machine. The index domain alignment technique presented in this dissertation finds a set of alignment functions that map the index domains of a set of arrays into a common index domain. The resulting alignment minimizes data movements that are caused by the cross-references between the arrays. Our technique does not depend on a particular architecture, but on a class of machines that are modeled by a reference metric. The evolution of machines may change the metric somewhat, but the methodology will still apply.

Automatic generation of interprocessor communication is one of the most critical issues in compiling a high-level language for a distributed-memory machine. Our approach to the problem is based on analyzing program references and matching each reference with the most efficient communication routine. We have also developed algorithms for scheduling and synchronizing communication routines, which ensure that the compiler-generated communication preserves the data dependences of the original program and is deadlock-free.

Deriving parallel control structures from a functional program is a fundamental compilation issue. We have formulated the problem in terms of dependence vectors and dependence graphs and developed a control structure synthesis algorithm. Based on data dependence information, the algorithm automatically transforms a functional program into an explicit parallel program with various loop constructs.

We have designed a compiler model that uses all the techniques mentioned above. A compilation process consists of a sequence of program transformations: from a Crystal program to an aligned program, to a shared-memory program, to a partitioned program, and finally, to a message-passing program. We have also developed an optimization framework based on an abstract target machine and communication metrics, which consists of two levels of optimizations: intra-program-block (local) and

inter-program-block (global). With this framework, a compiler can pursue optimizations in a systematic way, rationally adjusting and balancing conflicting goals.

Compilation models and compilation techniques are not useful if they are not practical. We have implemented an experimental Crystal compiler for hypercube machines based on our compilation model. The compilation techniques presented in this dissertation have been incorporated into the compiler. The preliminary results that we have obtained on a set of application programs, ranging from Gaussian elimination to a financial application, show that the compiler is able to generate code with performance within a factor of 1.7 to 2.8 of that of the corresponding hand-crafted programs.

8.2 Directions for Future Research

The work reported in this dissertation can be extended in many different directions. The following are some suggestions for future research.

Techniques for more dynamic problems We have shown that by restricting the arrays in the source program to be regular and static, the compiler is able to derive useful information from the program and generate efficient target code. For applications that need to use irregular data structures such as sparse matrices, compile-time analysis alone is no longer sufficient. Various *runtime analysis* techniques have been developed to handle dynamic problems (for instance, [SCMB90,KMSB90b]). More work needs to be done to see how to combine both compiler-time techniques and runtime techniques into a coherent framework.

User-compiler interaction Many large scientific applications require large programs with complicated structures. For these programs, using dependence-analysis based, fully-automatic techniques alone, the compiler may not be able to generate the most efficient code. A better approach is to provide tools for the user to interact with

the compiler, thus allowing the user to help the compiler make better decisions. With respect to the Crystal system, the user can provide concrete alignment functions, data layout strategies, communication forms, and so on.

Production-quality Crystal compiler The current experimental Crystal compiler has many limitations due to the shortage of manpower. Every part could be further improved. For instance, dependence analysis could be sharpened to handle conditionals; alignment techniques could be extended to cover more types of alignments; linear transformations could be added to the control structure synthesis module, and so on. In addition, new parts such as a performance estimator based on program profiling and a module for optimizing memory usage could also be developed. The Crystal group at Yale is now working towards building a production-quality Crystal compiler.

Applying the techniques to other languages The compilation techniques presented in this dissertation, though originally developed for Crystal, should be adaptable to other data-parallel languages such as Fortran 90. The array features of Fortran 90 are very suitable for our alignment techniques and communication generation techniques. Furthermore, the structure of the Crystal compiler has nice intermodule interfaces, making it very convenient to port to other source languages.

Bibliography

[AAG+87] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The WARP computer: Architecture, implementation, and performance. *IEEE Trans. on Computers*, C-36(12), December 1987.

[ABC+86] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, 1986. Springer-Verlag.

[AK87] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, 1987.

[Ban90] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* MIT Press, 1990.

[CC88] Y. Choo and M. Chen. A theory of parallel-program optimization. Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.

[CCH+88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: a parallel programming environment. *The International Journal of Supercomputing Applications*, 2(4):84–99, 1988.

[CCL88] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[Che86] M.C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, January 1986.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–170, 1988.

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Trans. on Programming Languages and Systems*, 8(4):491–523, October 1986.

[cmf89] *CM Fortran Reference Manual.* Cambridge, MA., version 5.2-0.6 edition, 1989.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[Ger89] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessor Systems.* PhD thesis, Bonn University, 1989.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability.* Freeman, San Fransico, CA, 1979.

[HJ88] C.-T. Ho and S.L. Johnsson. Stable dimension permutations on Boolean cubes. Technical Report YALEU/DCS/RR-617, Dept. of Computer Science, Yale University, October 1988.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. Technical Report TR90-154, Dept. of Computer Science, Rice University, March 1991.

[HKT91a] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. Technical Report TR90-156, Dept. of Computer Science, Rice University, April 1991.

[HKT91b] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-indenpendent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors.* Elsevier, (to appear) 1991.

[Hu91] Y. Hu. Experiments with asynchronous communications on iPSC/2. Technical Report In preparation, Dept. of Computer Science, Yale University, 1991.

[JH87] S.L. Johnsson and C.-T. Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. Technical Report YALEU/DCS/RR-610, Dept. of Computer Science, Yale University, November 1987.

[KKLW84] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced retargetable vectorizer. In K. Hwang, editor, *Tutorial on Supercomputers*, pages 163–178. IEEE Press, 1984.

[KLC+83] D.J. Kuck, D.H. Lawrie, R. Cytron, A. Sameh, and D.D. Gajski. The architecture and the programming of the Cedar system. Technical report, Laboratory for Advanced Supercomputers, Dept. of Computer Science, University of Illinois, August 1983.

[KLS88] K. Knobe, J. Lukas, and G. Steele, Jr. Massively parallel data optimization. In *Frontiers'88: The Second Symposium on athe Frontiers of Massively Parallel Computation*, Fairfax, VA, 1988.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[KM84] H.T. Kung and O. Menzilcioglu. Warp: A programmable systolic array processor. In *Proc. SPIE Symp. Real-Time Signal Processing VII*, 1984.

[KM89] C. Koelbel and P. Mehrotra. Compiler transformations for non-shared memory machines. In *Proceedings of the Forth International Conference on Supercomputing*, May 1989.

[KMSB90a] C. Koelbel, J. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KMSB90b] C. Koelbel, P. Mehrotra, J. Saltz, and H. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KMT91] K. Kennedy, K.S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, June 1991.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations.* Wiley, 1978.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Metroids.* Holt, Rinehart and Winston, 1976.

[LB80] S.F. Lundstrom and G.H. Barnes. A controllable MIMD architecture. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 19–27, 1980.

[LC90a] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[LC90b] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. In *Frontiers'90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC91a] J. Li and M. Chen. Compiling communication-efficient programs for massively-parallel machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3), July 1991.

[LC91b] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13, (to appear) 1991.

[LRS83] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference on VLSI*, pages 87–116. Caltech, March 1983.

[PKL80] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. on Computers*, C-29(9):763–776, September 1980.

[PW86] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), 1986.

[QHV88] M.J. Quinn, P.J. Hatcher, and J. Van Rosendale. Compiling C* programs for a hypercube multicomputer. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, CT, July 1988.

[RA82] J.A. Rees and N.I. Adams, IV. T: a dialect of Lisp or, LAMBDA: the ultimate software tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, August 1982.

[RAM88] J.A. Rees, N.I. Adams, and J.R. Meehan. *The T Manual*, fifth edition, October 1988.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[RS87] J.R. Rose and G.L. Steele Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[RS88] M. Rosing and R.B. Schnabel. An overview of DINO—a new language for numerical computation on distributed memory multiprocessors. Technical Report CU-CS-385-88, University of Colorado, Boulder, CO, March 1988.

[RSW88] M. Rosing, R.B. Schnabel, and R.P. Weaver. DINO: summary and examples. Technical Report CU-CS-386-88, University of Colorado, Boulder, CO, March 1988.

[RSW90] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, Boulder, CO, April 1990.

[Sar91] V. Sarkar. PTRAN—the IBM parallel translation system. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 8. ACM Press, 1991.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[SS78] G.L. Steele Jr. and G.J. Sussman. The revised report on Scheme, a dialect of Lisp. Technical report, AI Lab, MIT, Cambridge, MA, January 1978.

[Tse89] P.S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, May 1989.

[Tse90] P.S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[Ull84] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[WF91] M. Wu and G. Fox. Compiling Fortran90 programs for distributed memory MIMD parallel computers. Technical Report CRPC-TR91126, Center for Research on Parallel Computation, Syracuse University, January 1991.

[Wol82] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[Wol89] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. Technical Report ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[ZBG88] H.P. Zima, H.J. Bast, and M. Gerndt. Superb: A tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

Appendix A

A Compiler-Generated Program

Matrix Multiplication

```
/*
** Matrix Multiplication, Version mm2.1.
** Compiler generated.
** (The result() routine is hand-wrriten.)
**
*/

/*
** Node Program:
*/
#include <math.h>
#include COMP_HDR

#define n 64

int my_host,my_node,my_pid,rcnt,rnode,rpid,dest,num_nodes;
int cnt,type,size,unit_sz,bufsize1,bufsize2,param[40];
int i,j,i0,jj0,i1,jj1,i2,j2,i20,j20;
int imin,jmin,imax,jmax;
int ia,ja,i_bz,j_bz,ni,nj;
float **a,**b,**c;
float **a_g1,**b_g1;
void *sendbuf,*recvbuf;
int *ibuf1,*ibuf2;
float *buf1,*buf2;
```

```
main()
{
  crecv(INIT,param,40);
  num_nodes = param[0];
  ni = param[1];
  nj = param[2];

  init_params();

  if (my_node>=num_nodes)
    exit();

  {
  compute_a();
  comm_a1();
  }
  {
  compute_b();
  comm_b1();
  }
  compute_c();

  result();

  csend(DONE,param,0,my_host,host_pid);
}


init_params()
{
  my_pid = mypid();
  my_node = mynode();
  my_host = myhost();
  unit_sz = sizeof(float);

  imin = 0;
  imax = (n-1);
  jmin = 0;
  jmax = (n-1);

  i_bz = (((imax-imin) /ni) +1);
  j_bz = (((jmax-jmin) /nj) +1);

  ia = (my_node/nj);
  ja = (my_node%nj);
```

```
  i0 = 0;
  jj0 = 0;
  i1 = block_ubound(imax,imin,ia,i_bz,ni,0);
  jj1 = block_ubound(jmax,jmin,ja,j_bz,nj,0);
  i20 = global_idx(imax,imin,ia,i_bz,ni,i0);
  j20 = global_idx(jmax,jmin,ja,j_bz,nj,jj0);

  a = alloc_float_2d_array(i_bz,j_bz);
  b = alloc_float_2d_array(i_bz,j_bz);
  c = alloc_float_2d_array(i_bz,j_bz);

  a_g1 = alloc_float_2d_array(i_bz,(jmax+1));
  b_g1 = alloc_float_2d_array((imax+1),j_bz);

  bufsize1 = (i_bz*j_bz);
  bufsize2 = max((i_bz* (j_bz*nj)),((i_bz*ni) *j_bz));
  sendbuf = alloc_float_1d_array(bufsize1);
  recvbuf = alloc_float_1d_array(bufsize2);
  buf1 = sendbuf;
  buf2 = recvbuf;
  ibuf1 = sendbuf;
  ibuf2 = recvbuf;
}

compute_a()
{
  int i,j;

  {
  i2 = i20;
  for ((i=i0); (i<i1); i++,i2++)
    {
    j2 = j20;
    for ((j=jj0); (j<jj1); j++,j2++)
      if (i2>=j2)
        a[i][j] = 1.0;
      else
        a[i][j] = 0.0;
    }
  }
}

comm_a1()
{
```

```
  int i,j;

  {
  cnt = 0;
  for ((j=jj0); (j<j_bz); j++)
    for ((i=i0); (i<i_bz); i++)
      buf1[cnt++] = a[i][j];
  float_row_bcast(buf1,buf2,cnt,nj,my_node,my_pid);
  cnt = 0;
  for ((j=jmin); (j<=jmax); j++)
    {
    if (is_dummy_idx(jmax,jmin,j_bz,nj,j) ==1)
      cnt = (cnt+i_bz);

    for ((i=i0); (i<i_bz); i++)
      a_g1[i][j] = buf2[cnt++];
    }
  }
}

compute_b()
{
  int i,j;

  i2 = i20;
  for ((i=i0); (i<i1); i++,i2++)
    {
    j2 = j20;
    for ((j=jj0); (j<jj1); j++,j2++)
      b[i][j] = 10.0*i2+j2;
    }
}

comm_b1()
{
  int i,j;

  {
  cnt = 0;
  for ((i=i0); (i<i_bz); i++)
    for ((j=jj0); (j<j_bz); j++)
      buf1[cnt++] = b[i][j];
  float_col_bcast(buf1,buf2,cnt,nj,ni,my_node,my_pid);
  cnt = 0;
  for ((i=imin); (i<=imax); i++)
```

```
    {
    if (is_dummy_idx(imax,imin,i_bz,ni,i) ==1)
      cnt = (cnt+j_bz);

    for ((j=jj0); (j<j_bz); j++)
      b_g1[i][j] = buf2[cnt++];
    }
  }
}

compute_c()
{
  int i,j;

  for ((i=i0); (i<i1); i++)
    for ((j=jj0); (j<jj1); j++)
      {
      int k;

      c[i][j] = 0;
      for ((k=0); (k<n); k++)
        c[i][j] = (c[i][j] + (a_g1[i][k] * b_g1[k][j]));
      }
}


result()
{
  int i,j;

  {
  for ((i=i0); (i<i1); i++)
    {
    printf("**N%d: ", my_node);
    for ((j=jj0); (j<jj1); j++)
      printf("%4.0f", c[i][j]);
    printf("\n");
    }
  }
}
```